MSc in CSTE
Small Scale Parallel Programming

Patricia Colbere
Cranfield University

March 2019

Contents

Abstract

We have developed a kernel that calculates a sparse matrix-vector product. Because it is a sparse matrix, it will be stored in a compressed format. The matrix can be stored in the CSR or ELLPACK formats so that it is much faster to get a result from the kernel. The matrices studied have been taken from matrix market and are of varying sizes so that we can get a better idea of the performances of our kernel. Since the ELLPACK format is less efficient, Some matrices are too big for it, which will we precise later. In order to have a fast kernel, we are using OpenMP and CUDA. However the vector will just be generated randomly with a size corresponding that of the matrix. We also have a serial version of the code as a reference to see how useful using threads and GPUs to parallelise our matrix-vector product is for our kernel. We have also made our design so that the code can easily be altered if necessary by deviding it into logical parts in multiple files. The program is tested for correctness as well as performances so that we know that our results are right and how long our product took for the kernel to finish. To get a more accurate measure of the performances of OpenMP and CUDA compared with the serial implementation, we are doing the product ten times for each of the test matrices. We will now explain how we will present this project in more details.

Chapter 1

Introduction

We have designed and implemented a kernel that can calculate the product of a sparse matrix and a vector. The matrix will be taken as an input when executing the project while the vector will be generated randomly with an appropriate size. The product is made when the matrix is stored in a compressed format, and to make it faster, it is made using OpenMP or CUDA. Therefore we have three implementations of our project: a serial one, one using OpenMP and one using CUDA. However, only the parts directly linked to the matrix-vector product differ.

We will first be presenting the design we employed. In order to understand it, we will explain the requirements for our kernel and our solution to it with the design we implemented in serial, OpenMP and CUDA. Then we will show our test plan, which tests our kernel on both correctness and performances. Finally, we will present our performance analysis for the OpenMP and the CUDA implementations compared with the serial implementation.

Chapter 2

Design of our kernel

To explain why we chose to design our solution the way we did, we will first explain the requirements we answered to before presenting the design we implemented in more details. Most of the code is shared between our different implementations so we will be presenting that part before presenting the differences respective to our serial, OpenMP and CUDA implementations of the kernel.

2.1 Requirements of the kernel

The main functionality of our kernel is to get the result of a sparse matrix-vector product of the form y $\leftarrow$ Ax. Here A is stored in CSR or ELLPACK. The computing of the result has to be done in serial, OpenMP or CUDA. It was crucial to implement a first version of the kernel that was the most basic possible, so we began by implementing the matrix-vector product without any compression in the storage of the matrix and a serial implementation of the product. However, we were using thirty test matrices from matrix market, so we also needed to implement the code necessary to read the matrix. Our program asks the user to input the path to the matrix market matrix in the arguments of the command to execute the kernel. If the user does not input a path, we chose by default the cage4 matrix which is a little matrix of size 9x9 so that the program can execute quickly.

However, most matrices are much bigger, like the thermal2 matrix which has a size of 1228045x1228045, and cannot be computed that way. Therefore what we implemented next was the compression in CSR and ELLPACK. Indeed, the matrices from matrix market are compressed in COO with three matrices that contain each the values of the rows containing a non zero, that of the columns containing a non zero and the values of the non zeroes. Their size is the number of the non zero values. Because we wanted to do our product with A stored in CSR and ELLPACK, we created a file containing the necessary functions for a compression in CSR and for a compression in ELLPACK of a matrix already stored in COO.

Even so, the computations were much too slow for the large matrices, so we needed to parallelise the product. Therefore, we needed to change our implementation of the product itself as well as the way it is called in the main function. Indeed, for the OpenMP implementation, we had to use the different threads available so that the solution is given by multiple threads. For the CUDA imlementation, we made our product functions global and made use of the multiple GPUs of the device to calculate at most one row of the result per thread.

Moreover, we needed to be able to test the performances and the correctness of our kernel, so we made a test plan for that. We created a file containing the tests for the correctness of every function we implemented so that overall the reading of a file, the compression of its matrix in CSR or ELLPACK and the result given by the matrix-vector product are all correct in serial, OpenMP and CUDA. This way, we know that the correctness of our program is good. We also needed to test the performances of our program. Because we are testing the performances of our product calculations, we only measured the time it took for our kernel to do this calculation on our test matrices in every possible condition. We also only tried to improve the performances of this product and not that of the compression for instance, which was not directly relevant here. Therefore we timed the time it took for our kernel to solve the matrx-vector product and tried to improve that in order to get correct performances.

Now that we have explained how we interpreted the requirements, we will show in more details how we implemented our code in serial, OpenMP and CUDA and the necessary differences between them.

2.2 Design employed

We have three different implementations of our kernel: a serial one, one with OpenMP and one with CUDA. However, as we have explained, only the parts related directly to the computation of the product and the main differ, so most of our code remains the same for all three implementations. Therefore we will begin by presenting this design shared by our implementation and then we will show how our serial, OpenMP and CUDA versions are implemented on the parts where they are different.

2.2.1 Shared design

Most of the implementation for the three versions of the kernel is the same, so we will concentrate on this part for now. First, we will show the class diagram of our serial version to get a clearer understanding of this design, though we will only explain the parts unique to the serial version in the next section.

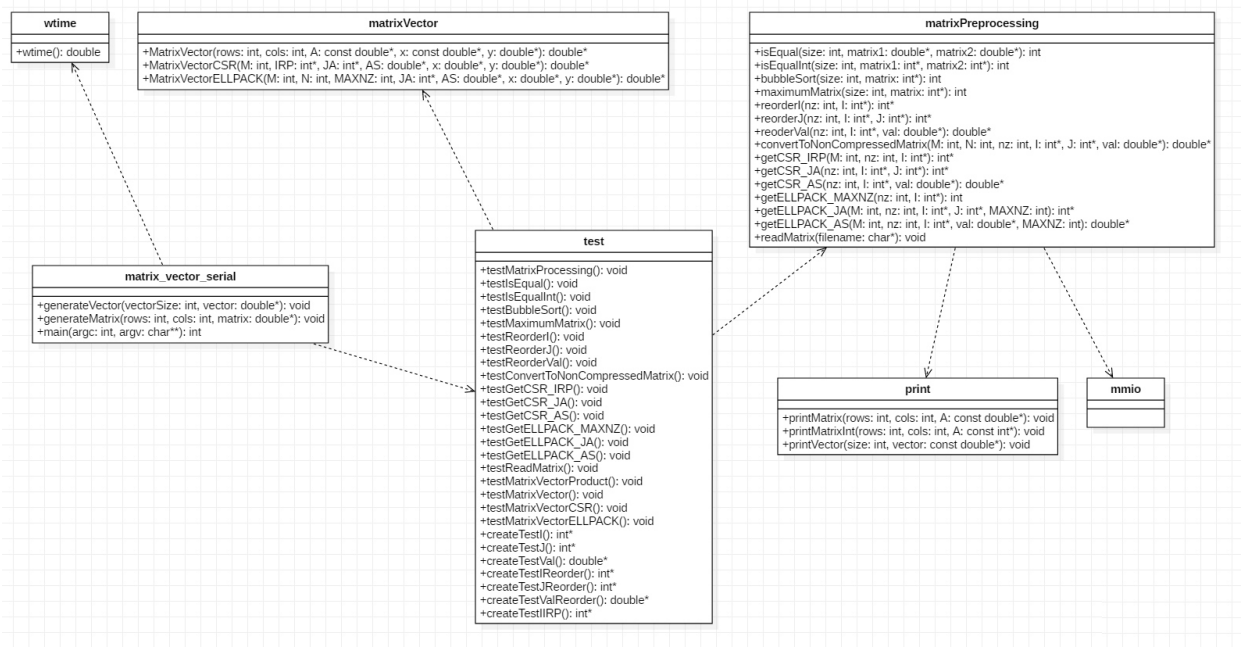

Figure 2.1 UML class diagram of the serial implementation

As we can see we have seven files in our project. These files remain the same for every implementation of our kernel. Only the main in martix_vector_serial (in the other versions matrix_vector_OpenMP and matrix_vector_CUDA) changes with each version, as well as the functions of the matrixVector file. Because we only implemented the MatrixVector function for a non compressed matrix to get a first version of our kernel, it has not been parallelised so it is also the same for each version.

We will now explain the contents of each file in more details. The actual contents of every file will be submitted alongside this report, ready to be executed for the three versions of our kernel. There are two files we have recuperated without changing them: wtime, which we use to measure the time take by our matrix-vector product and our performances, that we got from the solutions folder of the OpenMP exercises; and mmio, which we use to read the matrix from the file input, that we got from the matrix market website to read their matrices.

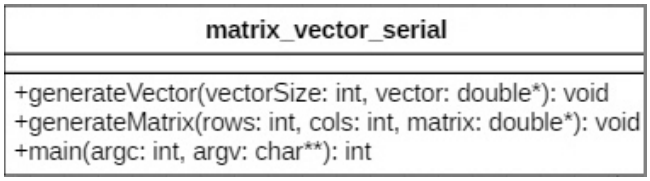

Figure 2.2 Diagram of the matrix_vector file

The file containing the main is matrix_vector. It has the same diagram in all of our three implementations, as it is only the contents of main which change.

We have a generateVector function which we use to generate a random vector x of the size needed to compute y ← Ax on our kernel. The generateMatrix function was used in our most basic version of the kernel where the matrix A was not read from a matrix market file but directly generated randomly. The main function is the one that calls the tests for the correctness and performances of our program, that reads the file input in the command, that compresses it into a matrix of the format CSR or ELLPACK and that computes the matrix vector product to show us the performances of our kernel. Only the way it calls the product and shows the performances is different between our implementations.

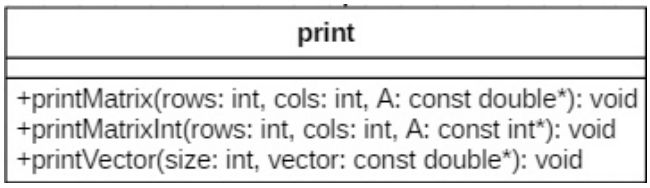

Figure 2.3 Diagram of the print file

This file contains the functions we used to check visually the contents of our vectors and matrices. It contains a function printMatrix that prints to the output the contents of a matrix of a given size if it contains double values. The function printMatrixInt has the same effect for a matrix containing integer values. Finally, the printVector function shows the contents of a given vector with double values. We have not implemented the equivalent for a vector with integer values because the only vectors we handle are x for the vector used in the Ax product and y for the vector containing the result of the product, and they both contain double values.

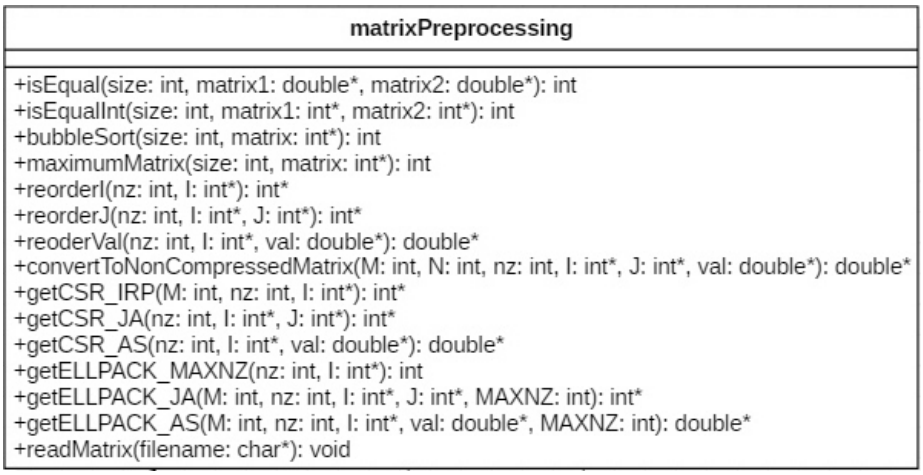

Figure 2.4 Diagram of the matrixPreprocessing file

The matrix preprocessing file contains every step necessary to go from a matrix written in the COO format used by matrix market to a CSR or an ELLPACK format. The function isEqual is used to check the correctness of the results returned by various functions: it tells us if the two matrices input of the same

given size have the same values, with the two matrices containing double values. The function isEqualInt does the same for two matrices with integer values. bubbleSort is used to sort a matrix of a given size by increasing order, with the matrix containing integer values. It is useful to transform the I, J and val matrices. Indeed, the way we get them from the reading example of matrix market that we use in the main, the matrices are all sorted in column-major order. However, since we are coding in C and later in CUDA, we use our matrices in row-major. That means that we need to transform these three matrices to row-major. For this, we use the functions reorderI, reorderJ and reorderVal that each use a bubbleSort of I and sort J and val the same way, so that the matrices are in row-major order and we can use them later on.

For the first basic implementation of our kernel, we used a matrix in a non compressed format to do our matrix vector product. However, we get the matrix in a COO format. That is why we made the function convertToNonCompressedMatrix that takes the matrix in COO format and returns a non compressed matrix. Once we had verified that this worked, we could directly convert the COO matrix to CSR and ELLPACK. In order to get the appropriate variables, we implemented the functions getCSR_IRP, getCSR_JA and getCSR_AS to get the matrices representing our A in the product when it is compressed to the CSR format. We also implemented getELLPACK_MAXNZ, getELLPACK_JA and getELLPACK_AS to get the value and the matrices representing A when compressed to the ELLPACK format. Finally, we have implemented in this file a readMatrix function so that is can be tested, that reads a matrix from a given file and gets its corresponding matrices for a COO format.

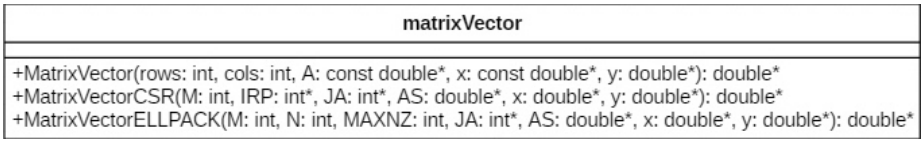

Figure 2.5 Diagram of the matrixVector file

Our matrixVector file contains the functions that calculate the matrix-vector product. As we have explained earlier, we first needed a function to do this calculation for a non compressed matrix, which is the function MatrixVector. Once that was done, we could do the functions MatrixVectorCSR and MatrixVectorELLPACK that do the calculation for a matrix compressed in CSR and ELLPACK respectively. Therefore, they take as an input the variables corresponding to the compressed matrices, as we will show in more details in the next section.

We also have a test file, but we will explain its use in the next chapter, as it is crucial to our test plan of the correctness of our implementation

We will now see the parts of our design unique to one implementation.

2.2.2 Design of the serial implementation

For our serial implementation, we have a basic non parallelised implementation of our matrix-vector product executed by the CPU with a single thread. We time how long it takes to execute and get the result in the main. Here is our code for the call of the product with the performances measured, followed by the implementation of this product in serial for a matrix compressed in CSR and ELLPACK. Because the performances are measured with an average to be more exact, we call the functions ten times each and we calculate the average of these ten times to get our performances, in seconds and in MFLOPS here.

Listing 2.1 Call to the function of the product in CSR

```
double t1CSR = wtime();
for (int i=0;i<10;i++)
    MatrixVectorCSR(M, IRP, JA_CSR, AS_CSR, x, y);
double t2CSR = wtime();
```

Listing 2.2 Call to the function of the product in ELLPACK

```
double t1ELLPACK = wtime();
for (int i = 0;i < 10;i++)
        MatrixVectorELLPACK(M, N, MAXNZ, JA_ELLPACK, AS_ELLPACK, x, y);
double t2ELLPACK = wtime();
```

Listing 2.3 Implementation of the serial product in CSR

```
// Implementation of matrix-vector product with the matrix in CSR
double* MatrixVectorCSR(int M, int* IRP, int* JA, double* AS, double* x,
double* y) {
        double temp;
        for (int i = 0;i < M;i++) {
                temp = 0;
                for (int j = IRP[i];j <= IRP[i + 1] - 1;j++) {
                        temp += AS[j] * x[JA[j]];
                }
                y[i] = temp;
        }
        return y;
}
```

Listing 2.4 Implementation of the serial product in ELLPACK

```
// Implementation of matrix-vector product with the matrix in ELLPACK
double* MatrixVectorELLPACK(int M, int N, int MAXNZ, int* JA, double* AS,
double* x, double* y) {
        double temp;
        int idx; // The index of (i,j)
        for (int i = 0;i < M;i++) {
                temp = 0;
                for (int j = 0;j < MAXNZ;j++) {
                        idx = i * MAXNZ + j; // The size of JA is M * MAXNZ
                        temp += AS[idx] * x[JA[idx]];
                }
                y[i] = temp;
        }
        return y;
}
```

As we can see, the implementations in CSR and ELLPACK use the compressed formats to calculate the product as was shown in the assignment pdf, adapted because our indices start at zero. Indeed, for IRP and the two JA matrices, their contents consider the indices starting at zero.

2.2.3 Design of the OpenMP implementation

For our OpenMP implementation, the only difference in the main is that we do our printing only with the main thread, once every thread has been syncronised, to get the correct results concerning the performance of our OpenMP implementation. However, our implementation of the matrix-vector product itself is different from the serial one. We will present it now.

Listing 2.5 Implementation of the OpenMP product in CSR

```
// Implementation of matrix-vector product with the matrix in CSR
double* MatrixVectorCSR(int M, int* IRP, int* JA, double* AS, double* x,
double* y) {
        double temp;
        int i, j;
        // We try to unroll to 4 if possible
#pragma omp parallel for shared(x,y,IRP, JA, AS) private(i,j,temp)
        for (int i = 0;i < M;i++) {
                temp = 0;
                if (IRP[i + 1] - IRP[i] >= 4) {
                        for (int j = IRP[i];j <= (IRP[i + 1] - 1) -
                        (IRP[i + 1] - 1) % 4;j += 4) {
                                temp += AS[j] * x[JA[j]] + AS[j + 1] *
                                x[JA[j + 1]] + AS[j + 2] * x[JA[j + 2]] +
                                AS[j + 3] * x[JA[j + 3]];
                        }
                        for (int j = (IRP[i + 1] - 1) - (IRP[i + 1] - 1) % 4;
                        j <= IRP[i + 1] - 1;j++) {
                                temp += AS[j] * x[JA[j]];
                        }
                }
                else {
                        for (int j = IRP[i];j <= IRP[i + 1] - 1;j++) {
                                temp += AS[j] * x[JA[j]];
                        }
                }
                y[i] = temp;

        }
        return y;
}
```

For the CSR implementation in OpenMP, we have chosen to unroll the calculation to four so that the access to memory uses more cache and is therefore more efficient. Unrolling more than to four did not seem very useful so we kept it at this value. The for loop is parallelised so that multiple threads can work on it at the same time.

Listing 2.6 Implementation of the OpenMP product in ELLPACK

```
// Implementation of matrix-vector product with the matrix in ELLPACK
double* MatrixVectorELLPACK(int M, int N, int MAXNZ, int* JA, double* AS,
double* x, double* y) {
        double t, t0, t1, t2, t3;
        int i, j, idx; // idx is the index of (i,j)

        // We unroll to 4 to reduce the loading time
#pragma omp parallel for shared(x,y,MAXNZ, JA, AS,M,N) private(i,j,idx,t0, t1, t2)
        for (i = 0;i < M - M % 4;i += 4) {
                t0 = 0;
                t1 = 0;
                t2 = 0;
                t3 = 0;

                for (j = 0;j < MAXNZ - MAXNZ % 2;j += 2) {
                        t0 += AS[(i + 0)*MAXNZ + j + 0] * x[JA[(i + 0)*MAXNZ +
                        j + 0]] + AS[(i + 0)*MAXNZ + j + 1] * x[JA[(i + 0)*MAXNZ + j + 1]];
                        t1 += AS[(i + 1)*MAXNZ + j + 0] * x[JA[(i + 1)*MAXNZ +
                        j + 0]] + AS[(i + 1)*MAXNZ + j + 1] * x[JA[(i + 1)*MAXNZ + j + 1]];
                        t2 += AS[(i + 2)*MAXNZ + j + 0] * x[JA[(i + 2)*MAXNZ +
                        j + 0]] + AS[(i + 2)*MAXNZ + j + 1] * x[JA[(i + 2)*MAXNZ + j + 1]];
                        t3 += AS[(i + 3)*MAXNZ + j + 0] * x[JA[(i + 3)*MAXNZ +
                        j + 0]] + AS[(i + 3)*MAXNZ + j + 1] * x[JA[(i + 3)*MAXNZ + j + 1]];
                }

                for (j = MAXNZ - MAXNZ % 2;j < MAXNZ;j++) {
                        t0 += AS[(i + 0)*MAXNZ + j] * x[JA[(i + 0)*MAXNZ + j]];
                        t1 += AS[(i + 1)*MAXNZ + j] * x[JA[(i + 1)*MAXNZ + j]];
                        t2 += AS[(i + 2)*MAXNZ + j] * x[JA[(i + 2)*MAXNZ + j]];
                        t3 += AS[(i + 3)*MAXNZ + j] * x[JA[(i + 3)*MAXNZ + j]];
                }
                y[i + 0] = t0;
                y[i + 1] = t1;
                y[i + 2] = t2;
                y[i + 3] = t3;
        }

        for (i = M - M % 4;i < M;i++) {
                t = 0.0;
                for (j = 0;j < MAXNZ;j++) {
                        idx = i * MAXNZ + j;
                        t += AS[idx] * x[JA[idx]];
                }
                y[i] = t;
        }

        return y;
}
```

For the ELLPACK function, we unrolled to four as well, this time for the rows and the columns, so that the access to memory is once again faster and the whole product has better performances. Once again, the for loop is parallelised. We will now see how our CUDA implementation has been made.

2.2.4 Design of the CUDA implementation

For the CUDA implementation, there were more changes as it works a bit differently in the call of functions implemented to use the GPUs of the device. Therefore, a few preparations first had to be made in the main.

Listing 2.7 Changes in the main due to the CUDA implementation

```
// We create our CUDA matrices
double *d_AS_CSR, *d_AS_ELLPACK, *d_x, *d_y;
int *d_IRP, *d_JA_CSR, *d_JA_ELLPACK;
checkCudaErrors(cudaMalloc((void**)&d_AS_CSR, nz * sizeof(double)));
...
// Copy matrices from the host (CPU) to the device (GPU).
checkCudaErrors(cudaMemcpy(d_AS_CSR, AS_CSR, nz * sizeof(double),
cudaMemcpyHostToDevice));
...
// Calculate the dimension of the grid of blocks (1D)  necessary to cover
// all rows.
...
// Create the CUDA SDK timer.
...

// We do the product with CSR
timer->start();
for (int i=0;i<10;i++)
    MatrixVectorCSR << <GRID_DIM, BLOCK_DIM >> > (M, d_IRP, d_JA_CSR,
    d_AS_CSR, d_x, d_y);
checkCudaErrors(cudaDeviceSynchronize());
timer->stop();
...
// We do the product for ELLPACK
timer->reset();
timer->start();
for (int i = 0;i < 10;i++)
        MatrixVectorELLPACK << <GRID_DIM, BLOCK_DIM >> > (M, N, MAXNZ,
        d_JA_ELLPACK, d_AS_ELLPACK, d_x, d_y);
checkCudaErrors(cudaDeviceSynchronize());
timer->stop();
...
// We free the matrices and vectors
delete timer;
checkCudaErrors(cudaFree(d_IRP));
...
```

As we can see, because we are using the GPUs, we first needed to create the CUDA matrices corresponding to the compressed format of A and to x and y. Then we get the values they contain from the variables of the CPU. Once that is done, we can get the dimension of the grid of blocks appropriate for our matrix and solution sizes. We have decided to do it in 1D because it did not seem necessary to implement a 2D solution given the performances we had, as we will explain later. After that, we create a timer to measure the performances of our CUDA implementation. Then, we can finally call the functions of the product in CSR and ELLPACK, and synchronise the GPUs before getting the time taken that will give us our performances. Once the results have been shown, we can free every matrix and vector created previously, including the CUDA ones. We will now show our CUDA implementations of the product for a matrix compressed in CSR or ELLPACK.

Listing 2.8 CUDA implementation of the product in CSR

```
// Implementation of matrix-vector product with the matrix in CSR
// using one thread per row
__global__ void MatrixVectorCSR(int M, int* IRP, int* JA, double* AS,
double* x, double* y) {
        double temp;
        int tr = threadIdx.x;
        int i = blockIdx.x*blockDim.x + tr;
        if (i < M) {
                temp = 0;
                for (int j = IRP[i];j <= IRP[i + 1] - 1;j++) {
                        temp += AS[j] * x[JA[j]];
                }
                y[i] = temp;
        }
}
```

For the CSR implementation, since the value corresponding to the columns we go through varies, we only implemented our product with one thread per row of result. Therefore, different threads of our GPUs can work on different rows of the solution at the same time and the product is made faster.

Listing 2.9 CUDA implementation of the product in ELLPACK

```
// Implementation of matrix-vector product with the matrix in ELLPACK
// using a block of threads for each block of rows.
__global__ void MatrixVectorELLPACK(int M, int N, int MAXNZ, int* JA,
double* AS, double* x, double* y) {
        __shared__ double ax[16][64];
        double temp;
        int tr     = threadIdx.y;
  int tc     = threadIdx.x;
  int i    = blockIdx.x*blockDim.y + tr;
  ax[tr][tc] = 0.0;
        if (i < M) {
                int idx = i * MAXNZ + tc;
                temp = 0;
                int j;
                for (j = tc;j < MAXNZ;j+=64) {
                        temp += AS[idx] * x[JA[idx]];
                        idx+=64; // The size of JA is M * MAXNZ
                }
                if (j<MAXNZ){
                        temp += AS[idx] * x[JA[idx]];
                }
                ax[tr][tc] = temp;
        }
        __syncthreads();
        for (int s=64/2; s >32; s >>=1){
                if (tc<s)
                        ax[tr][tc] += ax[tr][tc+s];
                        __syncthreads();
        }
        for (int s=min(32,64/2); s >0; s >>=1){
                if (tc<s)
                        ax[tr][tc] += ax[tr][tc+s];
        }

        if ((tc == 0)&&(i<M))
                y[i] = ax[tr][tc];
}
```

For our implementation with a matrix compressed in the ELLPACK format, we did a 1D block of threads for each block of rows of the result. This way, multiple threads of the GPUs can calculate a single line of our solution and we can get our final solution faster.

Therefore, we have seen that most of our implementation in serial, OpenMP and CUDA is the same, with a repartition of the types of tasks in different files: one for printing matrices and vector, one for reading the matrix market files, one to preprocess the matrix in CSR or ELLPACK, one to do the product, one to test the functions of the other files, one to time our calls to the product functions and the main to start the kernel containing all the other files. Amongst them, it is mostly the implementation of the matrix-vector product that differs, because the OpenMP uses the threads of the CPU to accelerate the computation of the product while CUDA uses the GPUs to get better performances. It will now be interesting to explain our test plan for the project, to test both its correctness and its performances.

Chapter 3

Test plan

For our project, we needed to test both the correctness of the result given by the kernel and its performances. We will present how we evaluated these two characteristics.

3.1 Testing the correctness of the project

Before trying to improve the performances of our kernel, we needed to make sure the results it gives us are correct. Therefore, we evaluated the functions implemented in our different files to verify that each step used by the kernel is correct and that the whole program works as expected. These verifications were made in the test file, that contains a function to check each functionality, and that is called in the main before the actual execution of the kernel using the command so that we know that the result given by the kernel is the right one. Therefore, we will now present our test file.

As is visible in the next figure, our test file contains two functions that call the other ones so that we only have to call these two functions in the main in order to test the whole program. Indeed, testMatrixProcessing tests every function in the matrixProcessing file while testMatrixVectorProduct tests the functions in the matrixVector file. For each function tested, we proceed the same way: we call the tested function to give us an expected result, and we compare the result obtained to the result expected. The test can be passed only if these two results are the same. This method is used for the serial, OpenMP and CUDA implementations. Therefore, the test file is exactly the same for the serial and OpenMP implementations. However, the CUDA implementation requires a void return type for its global functions, which is what we used for our implementations of the matrix-vector product, therefore with this method we could only test if the call is done correctly but the result itself cannot be checked this way, only manually. This is the way we verify every step we can to ensure that the whole process and the result given by our kernel for the product of the matrix input in the command line is the correct one.

Figure 3.1 Diagram of the test file

We have seen how we proceeded to make sure that our results are the right ones and that our kernel is correct. Now, we can focus on the performances of this kernel.

3.2 Testing the performances of the project

In order to make our project faster, we need to know its performances, so to measure them. Our way of measuring the performances is to time how long our program takes to do the matrix-vector product. Then, we convert this time to calculate the MFLOPS or GFLOPS taken by the kernel to find the solution. We will show how it is implemented.

Listing 3.1 Serial implementation of our test of performances

```
// We do the product with CSR
double t1CSR = wtime();
for (int i=0;i<10;i++)
        MatrixVectorCSR(M, IRP, JA_CSR, AS_CSR, x, y);
double t2CSR = wtime();
double tmltCSR = (t2CSR - t1CSR);
double mflopsCSR = (2.0e-6)*M*N*10 / tmltCSR;

// We do the product for ELLPACK
double t1ELLPACK = wtime();
for (int i = 0;i < 10;i++)
        MatrixVectorELLPACK(M, N, MAXNZ, JA_ELLPACK, AS_ELLPACK, x, y);
double t2ELLPACK = wtime();
double tmltELLPACK = (t2ELLPACK - t1ELLPACK);
double mflopsELLPACK = (2.0e-6)*M*N*10 / tmltELLPACK;

// We print our results
fprintf(stdout, "CSR: Matrix-Vector product of size %d x %d with 1 thread:
time %lf  MFLOPS %lf \n", M, N, (tmltCSR/10), mflopsCSR);
fprintf(stdout, "ELLPACK: Matrix-Vector product of size %d x %d with 1 thread:
time %lf  MFLOPS %lf \n", M, N, (tmltELLPACK/10), mflopsELLPACK);
```

For the serial implementation, we measure the time taken by MatrixVectorCSR and MatrixVectorELLPACK to calculate the product and show it to the user. Besides, since we want to get a more exact measure, we do the average of ten times taken by the calculation. Once that is done, we convert it from seconds to MFLOPS with the size of the matrix taken into account to have a more relevant measure of the performance of our kernel.

Listing 3.2 OpenMP implementation of our test of performances

```
// We print our results
#pragma omp parallel
                {
#pragma omp master
                        {
fprintf(stdout, "CSR: Matrix-Vector product of size %d x %d with %d threads:
time %lf  MFLOPS %lf \n", M, N, omp_get_num_threads(), (tmltCSR/10),
mflopsCSR);
fprintf(stdout, "ELLPACK: Matrix-Vector product of size %d x %d with %d
threads: time %lf  MFLOPS %lf \n", M, N, omp_get_num_threads(),
(tmltELLPACK/10), mflopsELLPACK);
                        }
                }
```

For the OpenMP implementation, we measure our performances exactly the same way with MFLOPS and it is only the way we show this to the user that changes, because only the master thread shows our results and it precises how many threads have been used.

Listing 3.3 CUDA implementation of our test of performances

```
// Create the CUDA SDK timer.
StopWatchInterface* timer = 0;
sdkCreateTimer(&timer);

// We do the product with CSR
timer->start();
for (int i=0;i<10;i++)
        MatrixVectorCSR << <GRID_DIM, BLOCK_DIM >> > (M, d_IRP, d_JA_CSR,
        d_AS_CSR, d_x, d_y);
checkCudaErrors(cudaDeviceSynchronize());
timer->stop();
double gpuflops = 10 * flopcnt / timer->getTime();

// We print our results
fprintf(stdout, "CSR: Matrix-Vector product of size %d x %d: time %lf
GFLOPS %lf \n", M, N, (timer->getTime())/10, gpuflops);


// We do the product for ELLPACK
timer->reset();
timer->start();
for (int i = 0;i < 10;i++)
        MatrixVectorELLPACK << <GRID_DIM, BLOCK_DIM >> > (M, N, MAXNZ,
        d_JA_ELLPACK, d_AS_ELLPACK, d_x, d_y);
checkCudaErrors(cudaDeviceSynchronize());
timer->stop();
gpuflops = 10 * flopcnt / timer->getTime();

// We print our results
fprintf(stdout, "ELLPACK: Matrix-Vector product of size %d x %d: time %lf
GFLOPS %lf \n", M, N, (timer->getTime())/10, gpuflops);
```

For the CUDA implementation, we use a CUDA SDK timer and get our results of performances in GFLOPS, so that we can analyse them. Indeed, we measure our performance in order to improve our code after having analysed which part is the most time consuming and therefore crucial to improve.

Therefore, we have seen that we need to test our program for correctness, to make sure that the results we get and the whole process is right. Once that is done, we also need to improve the speed at which we obtain our results, and that comes from better performances, which we also test for our implementations in serial, OpenMP and CUDA. Now that we have seen how our measures are done, we can actually analyse the performances we measured to see which implementation is better, why, and what could be improved to make our kernel more efficient.

Chapter 4

Performance analysis

To analyse the performances of our OpenMP and CUDA implementations, we compare their results to those of the serial implementation. However, we have realised that what makes our program really slow is actually converting the matrix to CSR and ELLPACK rather than the matrix-vector product itself, which is what we tried to get good performances on. This means that we could not get the results of performances for every matrix because it took too long just to prepare the matrix before the product. For instance, for rdist2, which is only a 3198 x 3198 matrix with 56934 non zero values, it takes 56s to convert it to CSR and ELLPACK while it takes 0.001s to do the product for the CSR and ELLPACK matrices. Therefore, if more time was given for this project, it is definitely the efficiency of the compression to CSR and ELLPACK that should be improved in priority. Because of this, we could not identify the matrices that were too big for an ELLPACK compression. However, because our purpose was first to improve the performances of the product calculation, we will be studying the results we have for OpenMP and CUDA concerning these, which are the results of nine of the first ten test matrices for OpenMP and of the first ten test matrices for CUDA.

4.1 Performances of the OpenMP implementation

For the OpenMP implementation, first we needed to choose a number of threads to execute our code in the queue. Because we could not test the biggest matrices, the differences between different numbers of threads was not obvious. Indeed, it was only visible when using 32 threads or more instead of less. Therefore, we chose to do our executions with 32 threads.

To study our performances, we compare them with the serial implementation. In the next two figures are the values we measured for the CSR and ELLPACK implementations in serial and in OpenMP.

size of matrix	Serial CSR Performances (GFLOPS)	Serial ELLPACK Perfomances (GFLOPS)
9	*	*
416	*	*
765	*	*
1000	*	1.17045
1813	2	*
2021	*	0.597630727
2597	13.488818	*
3198	*	20.454408
16146	173.795544001	130.346658
62451	866.694978034	600.019600132

Figure 4.1 Values of the performance in GFLOPS for the serial implementation

size of matrix	OpenMP CSR Performances (GFLOPS)	OpenMP ELLPACK Perfomances (GFLOPS)
9	1.761E-06	0.000162
416	0.346112	0.346112
765	0.008481522	0.019187705
1000	2	2
1813	0.041087113	0.038670224
2021	0.059194797	0.059194797
2597	0.09991717	0.094327399
3198	0.156140519	0.156140519
16146	3.750982964	3.47591088

Figure 4.2 Values of the performance in GFLOPS for the OpenMP implementation

As we can see, we have measured the performances of our implementations in GFLOPS compared to the size of the matrix that was used by the kernel to do the matrix-vector product. Some of the times calculated with the serial implementation were too little to be measured so the performances in GFLOPS approximated to infinite, therefore we did the graph without these values. In the next two figures are the resulting graphs for CSR and ELLPACK formats of the matrix.

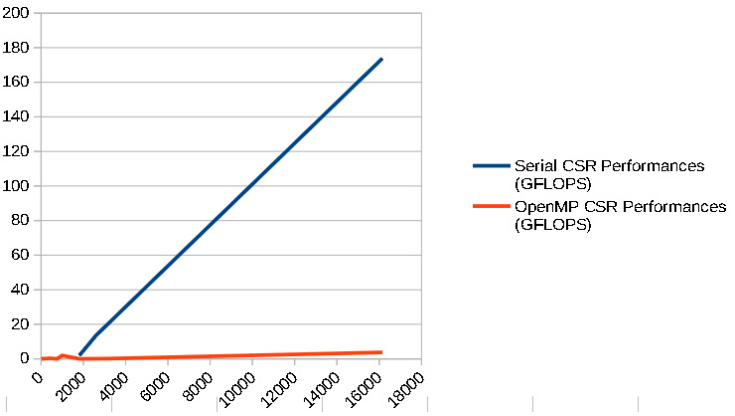

Figure 4.3 Graph of the number of GFLOPS for the size of the matrix for the CSR format, for the OpenMP and serial implementations

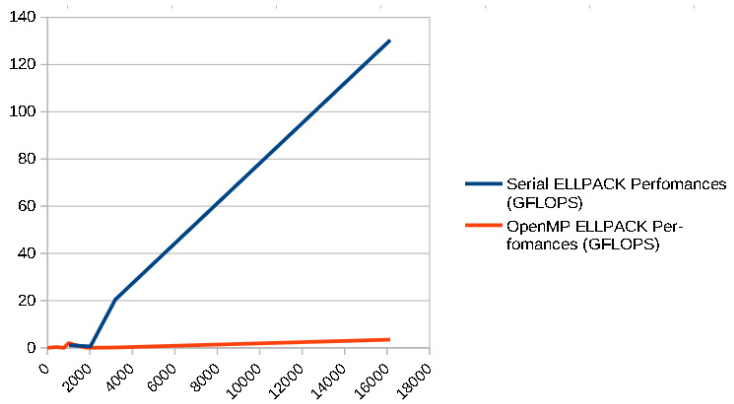

Figure 4.4 Graph of the number of GFLOPS for the size of the matrix for the ELLPACK format, for the OpenMP and serial implementations

As we can see, both serial and OpenMP get better performances with a matrix of a bigger size. However, the performances of the serial implementation grow much faster than those of the OpenMP implementation. This means that our access of memory is not well optimised and should be improved to get better performances for OpenMP, for instance by using cache memory more often than we do currently. It may also mean that the synchronisation of the threads takes too long and should be reduced more. Because OpenMP manages the threads implicitly, it also reduces the ways we can optimise our implementation compared to an explicit management of the threads. We should also probably take a better advantage of the pipelines by using blocks in addition to the threads we already use and by unrolling more of the code.

We will now study the performances our kernel has with its CUDA implementation.

4.2 Performances of the CUDA implementation

For the CUDA implementation, we proceeded the same way as for the OpenMP one. Indeed, we measured the performances in GFLOPS and compared them to those of the serial implementation. In the next figure are the values of our performances for the CUDA implementation of the kernel.

size of matrix	CUDA CSR Performances (GFLOPS)	CUDA ELLPACK Perfomances (GFLOPS)
9	0.014211	0.021316
416	8.360193	41.700243
765	15.102581	137.699999
1000	156.249993	259.740261
1813	23.630258	517.632897
2021	614.201648	1150.546723
2597	205.309253	1586.919748
3198	425.247562	2324.364558
16146	1326.008755	54311.107028
62451	4364.022917	577796.62904

Figure 4.5 Values of the performance in GFLOPS for the CUDA implementation

We can already see that these performances are much better than those of both the OpenMP and the serial implementations. Indeed, compared to the serial implementation, we have a performance 5 times bigger for the biggest matrix tested in CSR, the cant matrix of size 62451 x 62451, and more than 960 times bigger for this matrix in ELLPACK. This is probably because we only assigned one thread per row for the CSR implementation while we assigned a block of threads for a block of rows in ELLPACK, which means that we could use more threads in parallel and therefore get better performances. To visualise those results, the next figure contains the graph corresponding to our performances with CSR and ELLPACK respectively.

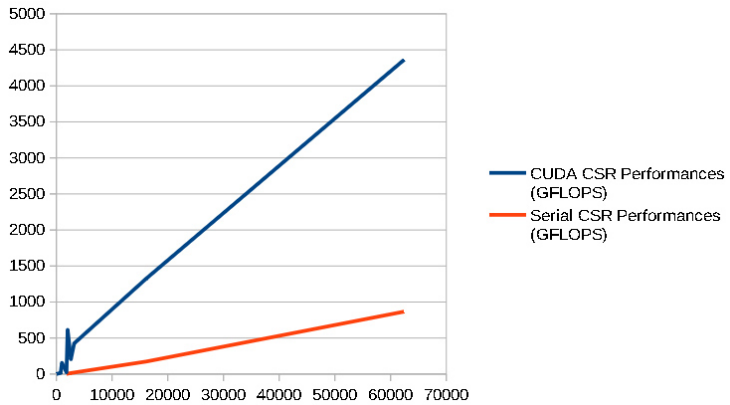

Figure 4.6 Graph of the number of GFLOPS for the size of the matrix for the CSR format, for the CUDA and serial implementations

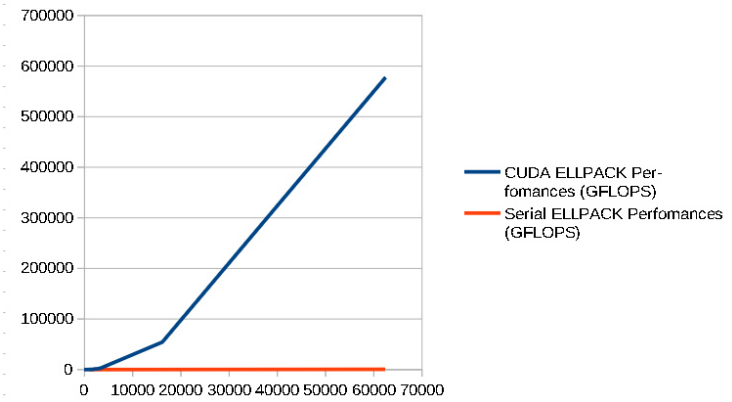

Figure 4.7 Graph of the number of GFLOPS for the size of the matrix for the ELLPACK format, for the CUDA and serial implementations

As can be seen, the performance of the CSR implementation is rather erratic despite our averaging of the results to calculate the performance. This may mean that our performances are really dependent on some characteristics of the matrix input in the kernel. This is something that should be corrected in order

to improve the performances of our CSR implementation in CUDA. However, the performance of our ELLPACK implementation seems almost exponential with the size of the matrix. This means that our ELLPACK implementation in CUDA is well optimised and we should concentrate on first improving the CSR implementation as well as the OpenMP implementation if we can. However, if we want to make the preprocessing faster in our CUDA implementation, we could have a full global memory access and process all of our vectors and matrices at the same time, because it is faster on GPU than CPU.

However, our biggest problem stays the preprocessing of the matrices, which we could also parallelise or directly include to the main to make it faster. Therefore, we have seen that while our performances in OpenMP were not optimised and actually worse than those of our serial program, the performances we obtained with our CUDA implementation were better than the serial ones, especially with an ELLPACK format of the matrix. This means that what should be improved in priority in our project is the preprocessing of the matrices, maybe by parallelising it, and the implementation of the matrix-vector product in OpenMP.

Chapter 5

Conclusion

We have presented the design of our project, which answers the requirements by having a common base to the serial, OpenMP and CUDA implementations and only having differences in the implementations of the main and of the calculation of the matrix-vector product of our kernel. Each of our implementations has the possibility to do the product with a matrix that is in the CSR format or the ELLPACK format, which are themselves obtained from a matrix file from matrix market. This implementation of our kernel is done thanks to various files that interact.

One of these files contains a series a tests for the correctness of our kernel result, by testing each step necessary to obtain a result and every function used in this process. Those tests are mostly the same for the serial, the OpenMP and the CUDA implementations. We also presented how we tested the performances of our kernel by timing how long our matrix-vector product takes and deducing the number of GFLOPS according to the size of the matrix in our product.

Finally, we analysed the performances we actually obtained for the OpenMP and the CUDA implementations compared to the serial one. We saw that our main problem concerning the performances was actually in our compression of the matrix to CSR and ELLPACK, though this is not linked to the performances of the product of the kernel, so it was not our main interest. Indeed, regarding our performances of the kernel, we saw that our CUDA implementation was much more efficient than our OpenMP implementation, which probably comes from a better use of the memory like the cache in our CUDA implementation.

Therefore, in order to improve the program, it would be a priority to get a more efficient compression of our matrix to CSR and ELLPACK and to improve our implementations of the matrix-vector product in OpenMP.

Chapter 6

Bibliography

- Course of Small Scale Parallel Programming by Salvatore Filippone
- Website of NVIDIA
- OpenMP API specification

Chapter 7

Appendices

7.1 How to use the project

For each implementation of our project, serial, with OpenMP and with CUDA, we have a folder containing all the files necessary. For all of them, only the matrix_vector_... file and the matrixVector files are different, but we still copied them to each folder for more convenience. Each of the folders also contains a sub file that enables the user to execute the program in a queue for every one of the thirty test matrices. If the sub files are not modified, a matrix folder containing the test matrices should be placed just above the given folders containing the sub files. The sub files give a half day to the jobs. The one for OpenMP also gives it 32 threads.