MI3.22 – Advanced Programming for HPC
Labwork 5 – *Project: matrix manipulation, minimum distance, sparse matrix*

Matrix is a very important tool in many different fields, from basic linear algebra, fluids mechanics, graph theory and applications, etc. This project aims at working with matrices using basic Cuda programming (so, no thrust here). Basic Cuda does not mean here simple solutions. On the contrary, the goal is to build robust and efficient algorithms, using all what you learnt during these two weeks, including shared memory, atomics, threads synchronization, among the others.

This project is made of three different distinct exercises, presented on three different pages:

1. In the first exercise, you will work with full matrix in order to compute minimum distances into a graph. This resembles a lot to the matrix product, even if the operation is not a multiply-addition, but a addition-minimum.
2. In the second exercise, the goal is to compute the determinant of a full matrix (with rank n). This implies to use the Gauss-Jordan algorithm, with pivoting ...
3. The last exercise aims at working with sparse matrices. Such matrices contain a lot of zeros. They are compressed by storing only the non zero values. Different ways to store the data exist, we choose here a simple one. Then, the goal is to build such a matrix, and then to do a matrix-vector multiplication (that produces a vector).

You cannot modify anything in the source code, except the `student?.cu` files. Your project should work efficiently using Cuda extensively. Solution made using the *host* but not the *device* will lead to mark 0. You may reuse piece of implementation seen into the course (like `DeviceBuffer`, `getSharedMemory`, `loadSharedMemory`, per-bloc reduction and or scan, etc.).

Exercise 1: Graph distance (5 points)

A directed graph is a structure made of vertices and directed edges (aka arcs). The arcs may have a valuation: a value that express for instance the cost to go from one vertex to another one.

A common way to store it uses a incidence matrix. It is square matrix of size $n \times n$, where for any vertex $v \in [0, n[$ the row v contains the arcs to all the other vertices. For instance, the graph containing 3 vertices and arcs $\{(0,1,4),(0,2,1),(1,2,1),(2,1,2)\}$, where the first number is the source vertex, the second the destination and the last the value, can be represented by the following matrix:

$$D = \begin{pmatrix} \infty & 4 & 1 \\ \infty & \infty & 1 \\ \infty & 2 & \infty \end{pmatrix}$$

The calculation of the minimum distance between every couples of vertices (v_1, v_2) can be computed thanks to many different algorithms. One solution consists to compute the incidence matrix exponent to n, with a particular product:

$$D = M^n$$

The product is not made here using a sum of products between row's elements and column' ones, but using the minimum distance formula: for a given cell $d_{i,j}$ of the product $D \times D$, the formula is then:

$$d_{i,j} = \min\left\{ d_{i,j}, \min_{k \in [0 \ldots n-1]} (d_{i,k} + d_{k,j}) \right\}$$

where the matrix D^1 is set to the incidence matrix.

Taking the previous graph example, then this product leads to the following distances (you should verify how it works, before to start coding ...):

$$D = \begin{pmatrix} \infty & 3 & 1 \\ \infty & 3 & 1 \\ \infty & 2 & 3 \end{pmatrix}$$

Question: Write the product D^n, using low-level Cuda only. It should be optimized with shared memory, synchronization etc. You may look at Internet for efficient classical matrix product, and then adapt to our problem ;-)

Notice that $D^n = D^m$ for any $m \geq n$. Then, the calculation should be done using $\lceil \log_2 n \rceil$ product only. You may take a look at the sequential solution to understand this point ...

Exercise 2: Determinant of a matrix (7 points)

Computing the determinant of a square matrix of size $n \times n$ is a quite important task for many different applications. Such a calculation is not obvious, nevertheless. A classical definition relies on the Leibniz formula:

$$\det(A) = \sum_{\sigma \in S_n} \left(\text{sgn}(\sigma) \prod_{i=0}^{n-1} a_{i,\sigma_i} \right)$$

but with a very high computational complexity (because the number of permutations in S_n of integers $[0, n[$ is $n!$ ... for instance with $n = 3$, you have six sequences $S_3 = \{(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1),(2,1,0)\}$).

A much more efficient solution relies on the Gauss-Jordan algorithm, also used for matrix inversion. Basically, this algorithm relies on three row operations (applied $n-1$ times):

1. Swap the positions of two consecutive rows (to find a non-zero diagonal value, called the pivot).
2. Multiply a row by a non-zero scalar (the pivot).
3. Add to one row a scalar multiple of another.

The last two steps compute for the pivoting row L_i and all the rows below (such that $j > i$), the new rows L_j:

$$L_j = L_j + \left(-\frac{L_{j,i}}{L_{i,i}} \right) L_i$$

Notice that, after finding the *good pivoting row*, all the values below the diagonal can be modified in parallel ...

The Gauss-Jordan algorithm allows to compute the determinant thanks to the following properties:

- Swapping two **consecutive** rows multiplies the determinant by -1 (rule for pivoting).
- Multiplying a row by a non-zero scalar multiplies the determinant by the same scalar (so we need to divide the result by this scalar).
- Adding to one row a scalar multiple of another does not change the determinant.

With this algorithm[1], the determinant is:

$$\det(A) = \frac{\prod \text{diag}(B)}{d}$$

where B is the row echelon matrix produced by Gauss-Jordan elimination, and d the product of the scalars by which the determinant has been multiplied (including the swapping rule).

It can be computed directly as $\det(A) = (-1)^p \prod \text{diag}(B)$ without modifying the pivoting row L_i, where p is the number of row **consecutive** permutations, as shown below:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -20 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -8 \end{vmatrix} = (-1)^0 \times (1) \times (-3) \times (-6) = 24$$

You may verify that the determinant is 24 using Leibniz formula: $1.5.1 - 1.6.8 - 2.4.1 + 2.7.6 + 3.4.8 - 3.7.5 = 24$.

You may also try to permute the last two rows after step 1, such that the last row becomes (0 0 4):

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -20 \end{vmatrix} = (-1)^1 \begin{vmatrix} 1 & 2 & 3 \\ \mathbf{0} & \mathbf{-6} & \mathbf{-20} \\ \mathbf{0} & \mathbf{-3} & \mathbf{-6} \end{vmatrix} = (-1)^1 \begin{vmatrix} 1 & 2 & 3 \\ 0 & -6 & -20 \\ 0 & 0 & 4 \end{vmatrix} = (-1)^1 \times (1) \times (-6) \times (4) = 24$$

Take care, σ should count all the consecutive rows' swapping. For example, if you permute the first and last row it is 3 consecutive swaps: the last two rows, then the first two, and then the last two:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{vmatrix} = (-1)^2 \begin{vmatrix} 7 & 8 & 1 \\ 1 & 2 & 4 \\ 4 & 5 & 6 \end{vmatrix} = (-1)^3 \begin{vmatrix} 7 & 8 & 1 \\ 4 & 5 & 6 \\ 1 & 2 & 4 \end{vmatrix} = (-1)^3 \begin{vmatrix} 7 & 8 & 1 \\ \mathbf{0} & \frac{3}{7} & \frac{38}{7} \\ 0 & \frac{6}{7} & \frac{20}{7} \end{vmatrix} = (-1)^3 \begin{vmatrix} 7 & 8 & 1 \\ \mathbf{0} & \frac{3}{7} & \frac{38}{7} \\ 0 & 0 & -8 \end{vmatrix} = 24$$

Question: Write a Cuda algorithm that computes the determinant of a square matrix of size $n \times n$. For this, use all the possible optimisations to obtain the most efficient solution (shared memory, synchronization ...). Notice that the calculation should use a temporary matrix (because the rows pivoting and elimination principle will modify the input!). Moreover, both for simplicity and efficiency you `may` use "*dynamic parallelism*" in Cuda (launching kernels into kernel!).

[1] Assuming that the determinant exists! If it does not, and denoting r the rank of the matrix, it will not be possible to find a non-zero pivot for the row $r+1$ ... If it occurs, then your solution should return 0.

Exercise 3: Product between a sparse matrix and a vector (8 points)

A sparse matrix is a matrix that contains a lot of zeros. Then, instead of storing all values as done into regular matrices, a sparse matrix only stores the non-zero values. This is a very important properties for many different applications.

For instance, in fluid simulations when you want to solve the Poisson equation onto a given regular 2-dimensional mesh, you need to solve a linear system $A \times X = B$ for a vector B of size n and A a square matrix of size $n \times n$, where n corresponds to the number of discrete elements in the mesh; a classical situation is then $n = 2^{20}$ elements, so the matrix contains 2^{40} elements! Fortunately, this matrix contains a lot of zeros! In fact, only the neighbors of the regular grid elements are non-zeros, meaning 4 values per row of the matrix! The number of non-zeros is (boundaries omitted) 4×2^{20}, so around 4 millions; that's a few, compared to the 2^{40} values of the full matrix! And this becomes even better with 3-dimensional regular mesh, with 2^{27} elements for instance (512 per dimension)!

Many solutions exist to store a sparse matrix. In this exercise we will use a very simple one, a mixture between COO (*COOrdinates list*) and CSR ()*Compressed Sparse Row*): for each non-zero value it stores the value, plus its column and row into three different vectors of equal length, being the number of non zero values. It also uses an extra *Row* vector to stores the start and end position of the rows (see below); its size is $n + 1$, where n is the number of rows in the matrix. Notice that in classical COO we do not store the lookup vector, while in CSR we do not store the row index of each value ...

While it allows to manipulate bigger matrices, sparse matrix representation leads to more complicated algorithms for doing things like sparse matrix inversion, sparse matrix product, sparse matrix per vector multiplication, sparse LU decomposition, sparse determinant calculation (which is almost always 0, except for diagonal or band matrices), and so on. *On ne peut pas tout avoir !*[2]

In this exercise, we just ask you to write two rather simple operations: *sparse matrix structure generation* and *sparse matrix per vector product*. As a bonus question, you may also try the product of two sparse matrices.

Notice that here matrices are not necessarily squared ones, and so we denote by m their number of rows, and by n their number of columns for matrices of size $m \times n$.

Question: Write two methods allowing to:

1. Generate a sparse matrix into the CSR representation. The input is a list of non zero values stored as a vector of structured elements, with three data members (value, row and column). The matrix structure also contains:
   - one vector for the non-zero values,
   - one vector for their column index,
   - one vector for their row index,
   - and one compressed row or *lookup* vector that says were each row starts into the three previous vectors. For instance, with a matrix of size 3×4, four values at position $(0, 0)$, $(1, 1)$, $(1, 2)$ and $(2, 3)$ where the first index represents the row and the last the column, then the lookup vector should be: $[0, 1, 3, 4]$.
     As you can see, this vector has length $m + 1$, so here 4 and not 3. This allows to store the length of the last row, as the extra value minus the starting position of the last row, so 4-3 = 1.
2. Multiply a sparse matrix per a given vector, for a matrix of size $m \times n$ and a vector of size n. The result is a vector of size m. You may try different solutions to optimize the calculation: one thread per row, one warp per row, a full segmented reduce, etc. Notice that here the matrix is sparse, but the vector is full. In others words, the vector is a classical `DeviceBuffer<T>` of size n ...
3. (optional) Multiply two sparse matrices of size $m \times n$ and $m' \times n'$ with $n = m'$, that returns a new sparse matrix of size $m \times n'$. The calculation may use the solution of exercise 1 (based onto the blocking pattern), with a strategy to handle the non zero values obtained in the resulting sparse matrix (because you are producing a new sparse matrix). Notice that you could use here some `thrust` functionality, for instance the `thrust::sort` method to sort the values (by their corresponding index into the full (without 0) result matrix, *id est* by `row*n' + col`). You may also use again atomic operations, to find a writing location of any non zero value (so, to store the value plus its row and column). To simplify you may do the dot product calculation twice into different passes:
   (a) A first product to compute the number of non zero produced values.
   (b) An allocation of the produced data (COO output).
   (c) A second calculation, with storage of the values.
   (d) Of course, you should generate a sparse matrix from that, so sort the values and calculate the lookup ...

[2] In french!