M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2024-2025)

TP4

Programmation GPU

OpenMP

julien.jaeger@cea.fr
adrien.roussel@cea.fr

The objectives of this practical work are:

- Implementing the computation offset model with OpenMP.

I Compilation & Verification

Environment:

In order to compile an OpenMP program with support for remote computation on GPU-type accelerators, both the compiler and the OpenMP runtime must support it. To do this, we need to load a special compiler environment with LLVM from the ROMEO supercomputer. To load the correct environment to use the installed `clang` compiler: `module load llvm/14.0.5-gcc_10.2.0-targets_X86_NVPTX`.

Compilation:

We will now compile our first OpenMP program, which will check that the "target" support is enabled for the NVIDIA GPUs available on Romeo. To do this, we need to add the following options to the command line: `-fopenmp-targets=nvptx64-nvidia-cuda -Xopenmp-target -march=sm_60`.

Q.1: The code in the `ONGPU` folder can be used to perform a simple test to check that GPU computation has been successfully deported. Create a `Makefile` file to take into account the above compilation information and check that execution has been successful. Please note that compilation only takes place on login nodes, and execution on a compute node with at least one GPU.

Q.2: Use the OpenMP runtime functions to display the number of devices available on the current system.

Q.3: Initiate a first team league and display the number of teams using runtime functions.

II Basic mathematical kernels

In this section, we'll look at programming elementary mathematical kernels such as `Axpy` and matrix multiplication (gemm).

Attention: To transfer data from/to the GPU using the map clause, if the data is a pointer, don't forget to specify the size of this data or you'll get an error

at runtime. If A is a vector and N is its size, proceed as follows:
`#pragma omp target map(to/from/tofrom: A[:N])`

Q.4: `Axpy` is a very common operation in linear algebra. Given x and y two vectors and a a real number, the operation `Axpy` performs the following calculation: $y = ax + y$. First, write a sequential program describing this operation. Then, using OpenMP directives, propose a parallelization on accelerator.

Hint: make the program as efficient as possible by taking advantage of the GPU's coalesced accesses, make sure that one team handles a contiguous portion of the vector. To do this, calculate a chunk of indices for each team, using the function `omp_get_num_teams()`.

Q.5: From the `gemm.c` file, perform the same exercise, but this time with a product between 2 matrices.

Hint: you can use the `collapse` clause in the `distribute` directive to distribute a set of loops to different teams.

III Sparse Matrix Vector kernel (SpMV)

The sparse vector-matrix multiplication kernel (SpMV) is one of the most widely used calculations in linear algebra and scientific applications. It involves multiplying a vector x by a matrix A such that $y = xA$. The matrix A is said to be *sparse*, meaning that it contains many zeros. Here, we want to explore different storage formats for the A matrix to gain in performance. Figures 1 and 2 show two formats used to compress the matrix compared with its naive representation.

The first format shown is the *Compressed Sparse Row* (CSR), or *Compressed Row Storage* (CRS), also known as the "Yale format". It consists of storing non-zero data using three arrays. Noting N the number of rows in the matrix, we have: an array with the column indices of each non-zero element, an array with the values of the non-zero elements and an array `row` of size $N+1$ corresponding to the indices in the first two arrays of the first non-zero element in each row. Thus, `row[N]` indicates the number of non-zero elements in the matrix and `row[i+1] - row[i]` indicates the number of non-zero elements in row `i`.

On the other hand, the Ellpack data format (nicknamed ELL) is based on a padding technique to limit the memory consumption of a hollow matrix compared with a hollow format, while favoring vectorized access to the data structure. In memory, the width of each matrix row is identical to match the maximum width of the starting matrix. If a row has fewer non-zero elements than the set width (i.e., the maximum number of non-zero elements per row), we pad this row with zero elements. This adds a little data and computation compared with a more conventional format such as CSR, but since the access pattern to this structure makes it easy to vectorize the code, one gains in efficiency.

Q.6: From the given program, write a function executing the calculation kernel **SpMV** with the CSR format, then deporting this function to the GPU. Do the results seem sufficient? If not, why not and how can you solve the problem?

Q.7: Taking the following example into account, write a function to convert a CSR matrix into Ellpack format.

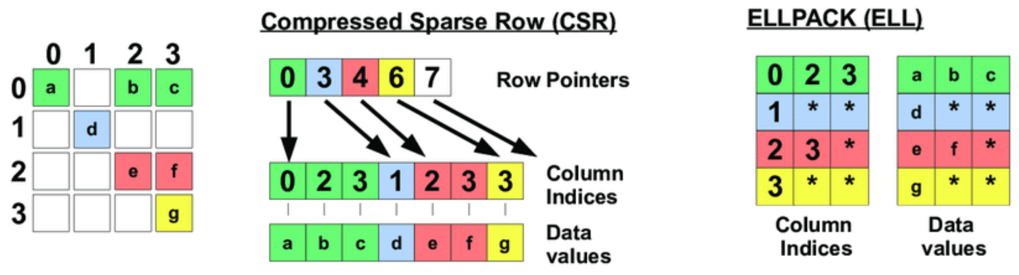
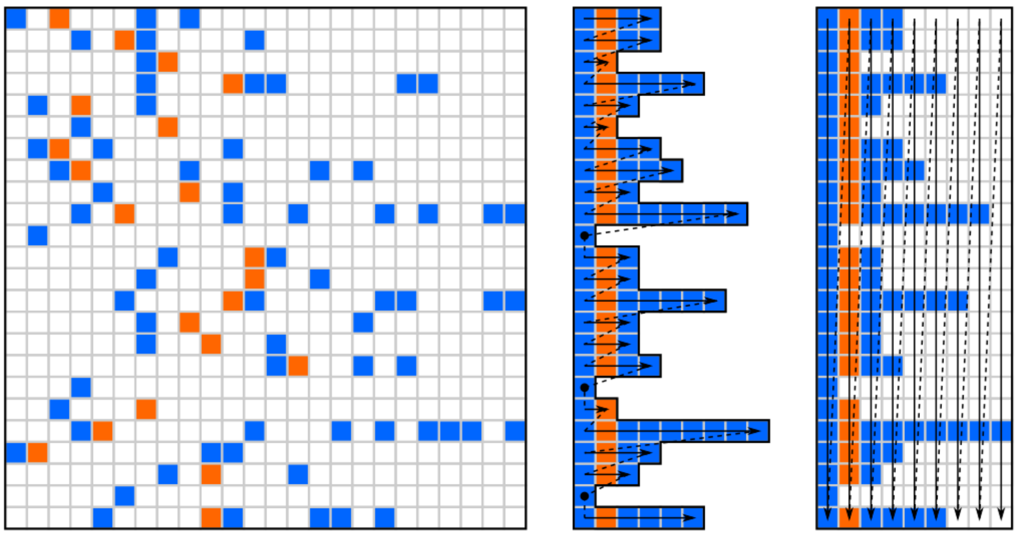


Figure 1: Two methods for compressing a sparse matrix: CSR et ELL



(a) Source matrix (b) CSR (c) ELLPACK

Figure 2: Memory access pattern for CSR and ELL formats. The arrows indicate the order in which matrix values and column indices are stored in memory. Non-zero orange elements form a column of elements in compressed formats. In CSR format, these are not stored in sequence.

Example : The matrix

$$A_{m,n} = \begin{pmatrix} a & 0 & 0 & b \\ 0 & c & 0 & 0 \\ 0 & d & e & 0 \\ f & 0 & g & h \end{pmatrix}$$

will have the following representation:

$$m\_values = \begin{bmatrix} a & b & 0 \\ c & 0 & 0 \\ d & e & 0 \\ f & g & h \end{bmatrix}$$

$$m\_cols = \begin{bmatrix} 0 & 3 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 3 \end{bmatrix}$$

Q.8: Based on this new format, propose a sequential code for the matrix-vector product (SpMV).

Q.9: Parallelize this new kernel with OpenMP offload support and target directives. Compare with previous results.