Sparse Matrix Multiplication

PUMPS 2015

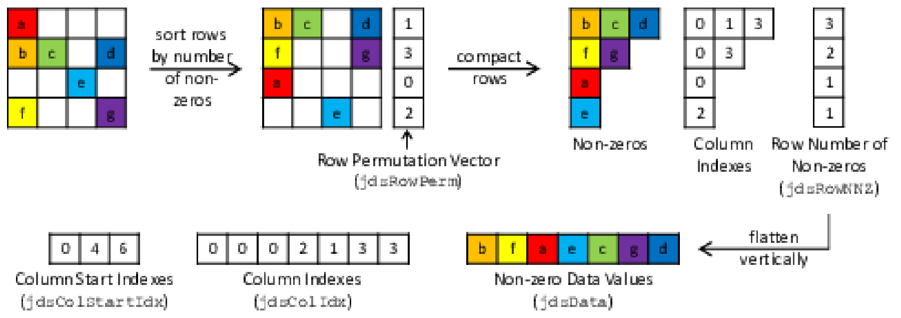
OBJECTIVE

The purpose of this lab is to understand sparse matrix storage formats and their impacts on performance using sparse matrix-vector multiplication as an example. The formats used in this lab are Compressed Sparse Row (CSR) and Jagged Diagonal Storage (JDS).

The following figure illustrates the components of a JDS matrix and how it is logically constructed.

INSTRUCTIONS

Edit the kernel and the host function in the file to implement sparse matrix-vector multiplication using the CSR format. Once complete, you can start working on the JDS format datasets.



sort rows by number of non-zeros
compact rows
Row Permutation Vector (jdsRowPerm)
Non-zeros
Column Indexes
Row Number of Non-zeros (jdsRowNNZ)
flatten vertically
Column Start Indexes (jdsColStartIdx)
Column Indexes (jdsColIdx)
Non-zero Data Values (jdsData)

Figure 1: image

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

QUESTIONS

- Consider an MxN sparse matrix with Z non-zeros and a maximum of k non-zeros per row. How much storage (in number of words) would a CSR representation of the matrix require? How much storage (in number of words) would a JDS representation of the matrix require?

- In the JDS format, what is the advantage of sorting rows according to the number of non-zero they contain?
- In the JDS format, what is the advantage of storing the data contiguously along the jagged diagonal instead of along the row?

SUGGESTIONS

- The system's autosave feature is not an excuse to not backup your code and answers to your questions regularly.
- If you have not done so already, read the tutorial
- Do not modify the template code provided – only insert code where the `//@@` demarcation is placed
- Develop your solution incrementally and test each version thoroughly before moving on to the next version
- Do not wait until the last minute to attempt the lab.
- If you get stuck with boundary conditions, grab a pen and paper. It is much easier to figure out the boundary conditions there.
- Implement the serial CPU version first, this will give you an understanding of the loops
- Get the first dataset working first. The datasets are ordered so the first one is the easiest to handle
- Make sure that your algorithm handles non-regular dimensional inputs (not square or multiples of 2). The slides may present the algorithm with nice inputs, since it minimizes the conditions. The datasets reflect different sizes of input that you are expected to handle
- Make sure that you test your program using all the datasets provided (the datasets can be selected using the dropdown next to the submission button)
- Check for errors: for example, when developing CUDA code, one can check for if the function call succeeded and print an error if not via the following macro:

```
#define wbCheck(stmt) do {                                          \
        cudaError_t err = stmt;                                     \
        if (err != cudaSuccess) {                                   \
            wbLog(ERROR, "Failed to run stmt ", #stmt);             \
            wbLog(ERROR, "CUDA error ...  ", cudaGetErrorString(err));\
            return -1;                                              \
        }                                                           \
    } while(0)
```

An example usage is `wbCheck(cudaMalloc(...))`. A similar macro can be developed while programming OpenCL code.

LOCAL DEVELOPMENT

While not required, the library used throughout the course can be downloaded from Github. The library does not depend on any external library (and should be cross platform), you can use `make` to generate the shared object file (further instructions are found on the Github page). Linking against the library would allow you to get similar behavior to the web interface (minus the imposed limitations). Once linked against the library, you can launch your program as follows:

```
./program -e <expected_output_file> \
    -i <input_file_1>,<input_file_2> -o <output_file> -t <type>
```

The `<expected_output_file>` and `<input_file_n>` are the input and output files provided in the dataset. The `<output_file>` is the location you'd like to place the output from your program. The `<type>` is the output file type: `vector`, `matrix`, or `image`. If an MP does not expect an input or output, then pass `none` as the parameter.

In case of issues or suggestions with the library, please report them through the issue tracker on Github.