GPU based optimizations for large graph adjacency matrices

Sanjana Garg, 13617

25 April, 2017

Introduction

BitMat is a novel compressed bit-vector based index structure previously developed for efficient storage of RDF graph's adjacency matrices. It stores the RDF triples using a 3-dimensional bit-matrix where the three dimensions are subject, object and predicate. A bit is set to 1 in this matrix if that triple is present. Since, this matrix is sparse, these mats are stored in a compresses format following the D-gap compression scheme.

Motivation

Fold and unfold are the two primitives in the BitMat system heavily used in the pattern queries for fast operations on the compressed matrices without uncompressing them. Fold and unfold operations operate on each compressed row in a "mutually exclusive" manner using mask bit-vectors (generated by pattern match queries). In this regard, the main contribution of this project is to exploit the "mutual exclusivity" of these operations using the parallelization offered by the GPUs.

Fold operation

The fold operation performs an **OR** operation over all the rows/columns to give a folded array of the column/row dimension. Consider an uncompressed bitmat of the following form. Here, the matrix represents the bitmat and the array below that represents the folded array.

Algorithm 1 CPU_Fold(BitMat, n, m)

1: **for** $i = 0$ to n **do**
2: **for** $j = 0$ to m **do**
3: $foldarr[j] \leftarrow foldarr[j]$ or $BitMat[i][j]$
4: **end for**
5: **end for**

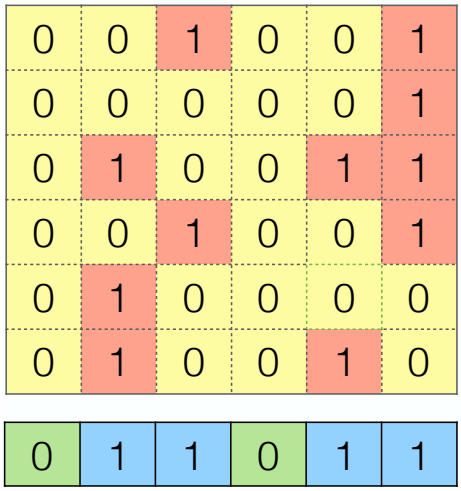

Figure 1: In this figure, matrix represents an uncompressed BitMat while the array represents the folded array.

Unfold operation

The unfold operation takes as input a bitmat and a mask array and outputs a bitmat with all it's rows ANDed with the mask array.

Algorithm 2 CPU_Fold(input, maskarr, output, n, m)

1: **for** $i = 0$ to n **do**
2: **for** $j = 0$ to m **do**
3: $output[i][j] \leftarrow maskarr[j] \& input[i][j]$
4: **end for**
5: **end for**

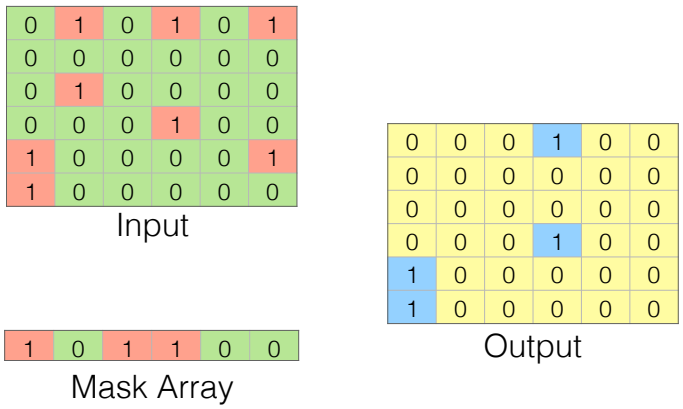

Figure 2: In this figure, matrix represents an uncompressed BitMat while the array represents the mask array. Output is the result of applying unfold operation on input as outlined in the algorithm above

Parallelization in CUDA

Architecture

CUDA is a library developed to exploit the parallelization in GPUs. The architecture divides the code and data into two parts: device and host. Here device, refers to the GPU and host to the CPU machine. The parallelization is performed by threads. The threads are organized into blocks where a block is executed by a multiprocessing unit. The blocks are further organized into a grid. Both the threads and blocks can have a 1, 2 or 3-dimensional index. The grids map to GPUs, blocks to multiprocessors and threads to stream processors.

In this project, I have used a 1-dimensional indexing for both threads and blocks as the parallelization is only on 1-dimension int the BitMat, either row or column.

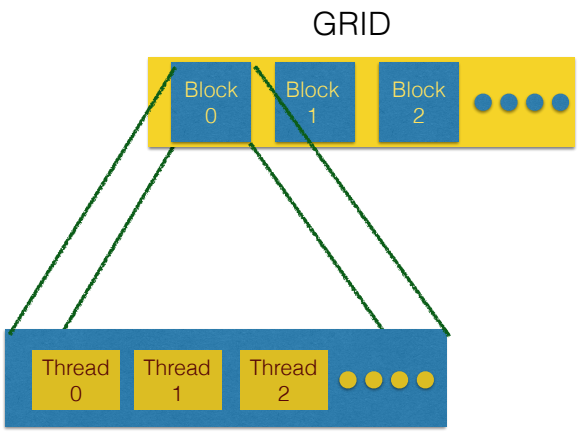

Figure 3: This figure represents the one-dimensional indexing based architecture for both threads and blocks

Configuration

Following are the hardware details of the machine used for experimentation

- GPU: NVIDIA GeForce GTX 760 2GB
- Maximum number of threads per block: 1024

Following is the configuration of threads and blocks employed in the project

$$
\begin{aligned}
n &= \text{Number of rows} \\
split &= \text{Number of rows assigned to a thread} \\
required\_threads &= ceil(n/split) \\
numBlocks &= ceil(required\_threads/threadsPerBlock)
\end{aligned}
$$

Parallelized Fold operation

Algorithm 3 GPU_Fold(bitmat,n,num_chunks, mask_size)

```
1: gpu_input_size ← get_size_of_gpu_input(bitmat)
2: split ← n/num_chunks
3: partial_rows ← ((n%split > 0)?(n/split + 1) : (n/split))
4: partial_size ← partial_rows ∗ mask_size
5: memset(partial,0,partial_size)
6: convert_bitmat_to_gpu_input(bitmat, gpu_input, mapping, n)
7: cudaMalloc(d_mapping, n∗ sizeof(int))
8: cudaMalloc(d_input, gpu_input_size ∗ sizeof(unsigned char))
9: cudaMalloc(d_partial, partial_size ∗ sizeof(unsigned char))
10: cudaMemcpy(d_mapping, mapping, n∗ sizeof(int) , cudaMemcpyHostToDevice)
11: cudaMemcpy(d_input, gpu_input, gpu_input_size ∗ sizeof(unsigned char), cudaMemcpyHostToDevice)
12: cudaMemcpy(d_partial, partial, partial_size ∗ sizeof(unsigned char), cudaMemcpyHostToDevice)
13: foldkernel⋘numBlocks,threadsPerBlock⋙(d_mapping, d_input, d_partial, n, mask_size, split)
14: cudaMemcpy(partial, d_partial, partial_size ∗ sizeof(unsigned char), cudaMemcpyDeviceToHost)
15: memset(foldarr,0,mask_size)
16: for i = 0 to partial_rows do
17:    for j = 0 to mask_size do
18:       foldarr[j] ← partial[i∗mask_size +j]
19:    end for
20: end for
```

Algorithm 4 foldkernel(d_mapping, d_input, d_partial, n, mask_size, split))

```
1: threadid ← blockIdx.x ∗ blockDim.x + threadIdx.x
2: start_row ← threadid ∗ split
3: for k = start_row; k < start_row + split; k++ do
4:    if k < n then
5:       data ← d_input + mapping[k]
6:       for j = 0; j < mask_size; j++ do
7:          partial[threadid∗mask_size+j] ← partial[threadid∗mask_size+j]
            | data[j]
8:       end for
9:    end if
10: end for
```