GPU based optimizations for large graph adjacency matrices

Sanjana Garg, 13617

28 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 ; i<n; i++) do
2:     for (j=0 ; j<m; j++) do
3:         $foldarr[j] \leftarrow foldarr[j]$ or $BitMat[i][j]$
4:     end for
5: end for
```

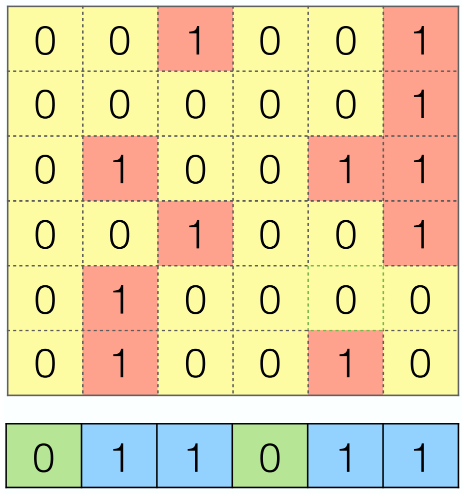

Figure 1: In this figure, matrix represents an uncompressed BitMat while the array represents the folded array.

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; i<n; i++) do
2:     for (j=0; j<m; j++) do
3:         $output[i][j] \leftarrow maskarr[j] \& input[i][j]$
4:     end for
5: end for
```

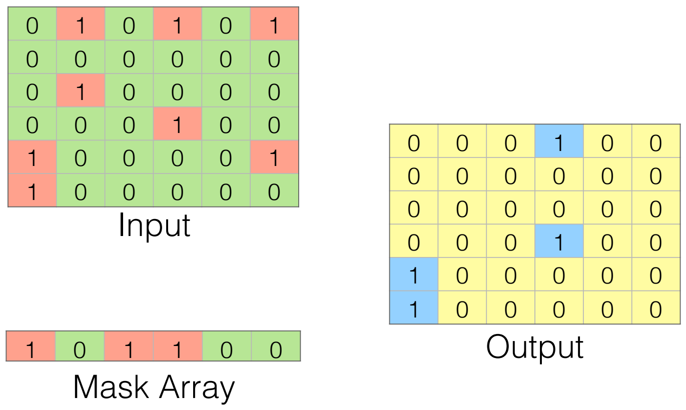

Figure 2: In this figure, matrix represents an uncompressed BitMat while the array represents the mask array. Output is the result of applying unfold operation on input as outlined in the algorithm above

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.

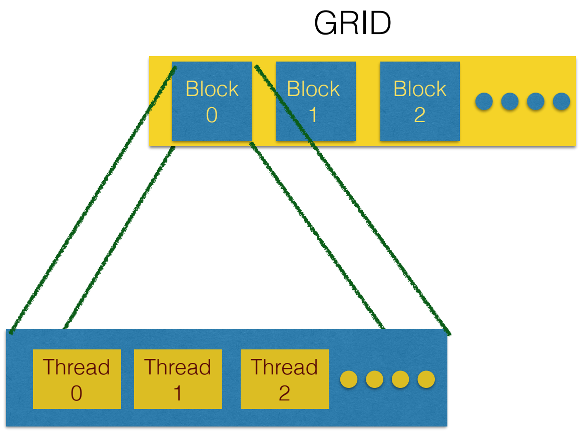

Figure 3: This figure represents the one-dimensional indexing based architecture for both threads and blocks

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)

```
gpu_input_size ← get_size_of_gpu_input(bitmat)
split ← n/num_chunks
partial_rows ← $((n\%split > 0)?(n/split + 1) : (n/split))$
partial_size ← partial_rows ∗ mask_size
memset(partial,0,partial_size)
convert_bitmat_to_gpu_input(bitmat, gpu_input, mapping, $n$)
cudaMalloc(d_mapping, $n*$ sizeof(int))
cudaMalloc(d_input, gpu_input_size ∗ sizeof(unsigned char))
cudaMalloc(d_partial, partial_size ∗ sizeof(unsigned char))
cudaMemcpy(d_mapping, mapping, $n*$ sizeof(int) , cudaMemcpyHostToDevice)
cudaMemcpy(d_input, gpu_input, gpu_input_size ∗ sizeof(unsigned char), cudaMemcpyHostToDevice)
cudaMemcpy(d_partial, partial, partial_size ∗ sizeof(unsigned char), cudaMemcpyHostToDevice)
foldkernel⋘numBlocks,threadsPerBlock⋙(d_mapping, d_input, d_partial, n, mask_size, split)
cudaMemcpy(partial, d_partial, partial_size ∗ sizeof(unsigned char), cudaMemcpyDeviceToHost)
memset(foldarr,0,mask_size)
for (i=0; i<partial_rows; i++) do
    for (j=0; j < mask_size; j++) do
        foldarr[$j$] ← partial[$i*$mask_size $+j$]
    end for
end for
```

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

```
threadid ← blockIdx.x ∗ blockDim.x + threadIdx.x
start_row ← threadid ∗ split
for (k=start_row; k < start_row + split; k++) do
    if $k < n$ then
        data ← d_input + d_mapping[$k$]
        for (j=0; j < mask_size; j++) do
            partial[threadid * mask_size + j] ← partial[threadid * mask_size + j] | data[$j$]
        end for
    end if
end for
```

Parallelized Unfold operation

Algorithm 5 GPU_UnFold(bitmat,n,mask_size)

```
1: gpu_input_size ← get_size_of_gpu_input(bitmat)
2: convert_bitmat_to_gpu_input(bitmat, gpu_input, mapping, $n$)
3: memset(gpu_output, 0, gpu_output_size)
4: cudaMalloc(d_mapping, $n*$ sizeof(int))
5: cudaMalloc(d_input, gpu_input_size $*$ sizeof(unsigned char))
6: cudaMalloc(d_mask, mask_size $*$ sizeof(unsigned char))
7: cudaMalloc(d_output, gpu_output_size $*$ sizeof(unsigned char))
8: cudaMemcpy(d_mapping, mapping, $n*$ sizeof(int) , cudaMemcpyHostToDe-
   vice)
9: cudaMemcpy(d_input, gpu_input, gpu_input_size $*$ sizeof(unsigned char),
   cudaMemcpyHostToDevice)
10: cudaMemcpy(d_mask, partial, mask_size $*$ sizeof(unsigned char), cudaMem-
    cpyHostToDevice)
11: cudaMemcpy(d_output, gpu_output, sizeof(unsigned char)$*$gpu_output_size,
    cudaMemcpyHostToDevice)
12: andres_size ← gap_size$*2*$ mask_size + gap_size +1 + 1024
13: unfoldkernel⋘numBlocks,threadsPerBlock⋙(d_mapping,        d_input,
    d_mask, d_output, n, andres_size)
14: cudaMemcpy(gpu_output, d_output, gpu_output_size$*$sizeof(unsigned char),
    cudaMemcpyDeviceToHost)
15: output_bitmat ← convert_output_to_bitmat(gpu_output,n, andres_size);
```

Algorithm 6 unfoldkernel(d_mapping, d_input, d_partial, n, mask_size, split))

```
1: row ← blockIdx.x $*$ blockDim.x + threadIdx.x
2: if (row < n) then
3:     if (d_mapping[row] < n) then
4:         return
5:     end if
6:     data ← d_input + d_mapping[row]
7:     o_data ← d_ouput + row $*$ andres_size
8:     for (j=0; j < mask_size; j++) do
9:         o_data[$j$] ← data[$j$] & mask[$j$]
10:    end for
11: end if
```

The above pseudocodes for Fold and Unfold operation on GPU do not include the complex logic for performing these operations in compressed format. However, the other framework and functions are same. This is done to keep the express the algorithm succinctly excluding the complications of compressed format. The variables used are self-explanatory and may not be declared in the pseudocode to maintain the preciseness.

Experiments

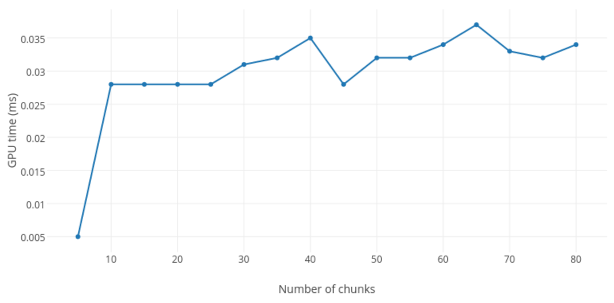

Figure 4: GPU operation time vs Number of chunks

It can be observed that changing the number of chunks (threads essentially) causes a very small effect in the time it takes for computation on GPU.

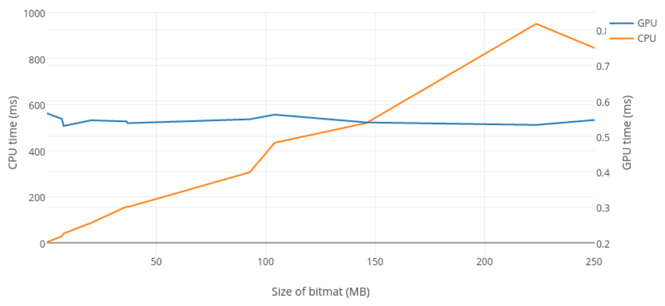

Figure 5: CPU/GPU time vs size of bitmat

Here, we observe that as the size of bitmat increases CPU time increases linearly whereas GPU time almost stays the same. This can be exploited for more efficient computation as size does not affect the GPU time computation as expected.

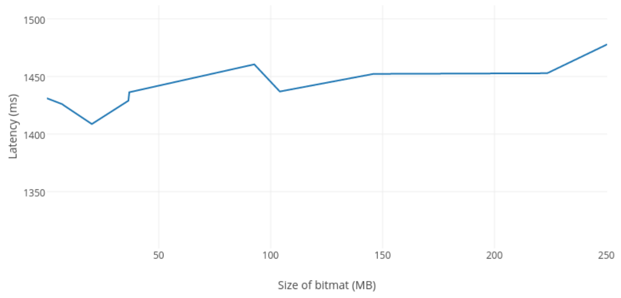

Figure 6: Latency due to data transfer vs size of bitmat

The above plot shows an interesting phenomena. Here, it can be observed that the latency time is almost constant for any size of data. Ideally, with increasing size memory transfer should take larger time. It actually reduces the transfer overhead as apparently the size of the bitmat does not affect it.

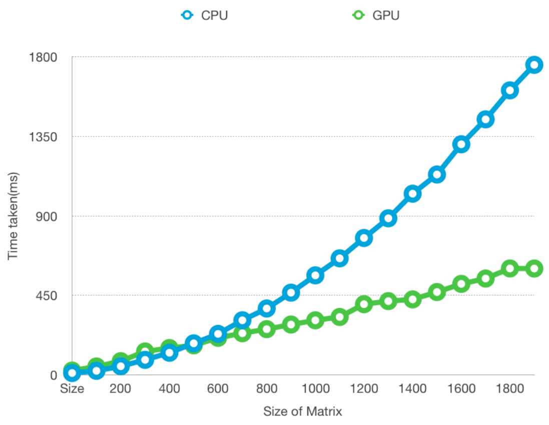

Figure 7: GPU/CPU time vs size of matrix

This plot shows the CPU and GPU performance while performing unfold operation. It can be observed that with increasing size of matrix the GPU performs considerably better than CPU.

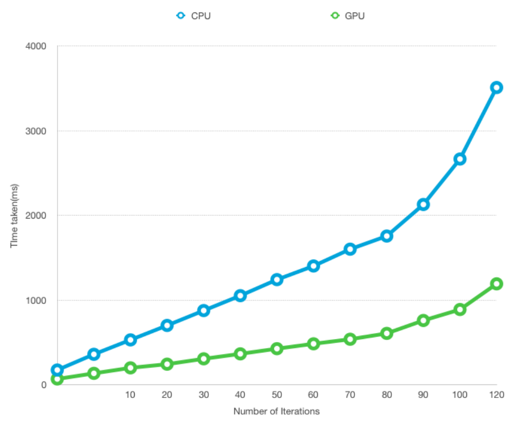

Figure 8: GPU/CPU time vs number of iterations

In the above plot, we observe that once data is there in the GPU memory the unfold operation on GPU takes considerably lesser time with frequent use. The performance difference between the GPU and CPU increases with the number of iterations.

Contribution

One of the major advantages of the above implementation for GPU is that the Fold and Unfold operations are performed in compressed format. This reduces not only the space complexity but as observed while doing operation on simulated data time complexity as well by a great amount.

Another plus point is that the implementation is compatible with the existing code base of BitMat project that performs pattern queries on these bitmats. This was also a major requirement as the existing code base has been perfected for quite a few years and it didn't make sense to rewrite the whole code for an operation. To accomplish this feat the bitmat struct was converted to a one-dimensional array with a mapping maintained as CUDA is too restrictive and doesn't support array of character arrays. In fact, one of the major bottleneck for CUDA implementation was transferring data to GPU memory and yet maintaining the compatibility.

The performance aim has been met in the above implementation especially for the unfold operation. We observe that GPU computation is significantly efficient even for small bitmats while doing unfold operation.

Future work

The future work includes restructuring the current code as a library so that it can be imported anywhere in the existing code base. Also, currently the Unfold operation could not be tested on actual dbpedia datasets because of memory issues. So, one of the major extension is a data structure that can function in the given memory constraints and perform computation on GPU.