1 Implementing ASD for big data in CUPY

Our goal in this article is to optimize the algorithm so that it is able to complete matrices too large to normally fit into memory. We still want to recover a matrix in LR form but from now on we will not actually compute the product LR in its entirety as it is unlikely to fit in working memory. Our very first challenge is that the implementation of createRandomSparseMask fails. This is because the code generates all the numbers between 1 and mn then shuffles them to get random indecies: we run out of memory allocating the integer array. We will instead generate the requested amount of indecies using a uniform sampling method and throw out any duplicates. As a result the sparse mask produced has slightly lower sparsity than requested but since this routine is used only to create test data it doesn't matter in practise.

```python
# produce a random sparse mask with specified dimensions and fill percentage
def createRandomSparseMask(m: int, n: int, density: float,
             random_state=None, dtype=cp.float32) ->cp.sparse:
    if density < 0 or density > 1:
        raise ValueError('density expected to be 0 <= density <= 1')
    dtype = cp.dtype(dtype)
    if dtype.char not in 'fd':
        raise NotImplementedError('type %s not supported' % dtype)
    if random_state is None:
        random_state = cp.random
    elif isinstance(random_state, (int, cp.integer)):
        random_state = cp.random.RandomState(random_state)
    # use numpy instead and remove duplicates
    ind = np.random.randint(low=0,high=n*m,size=int(n*m*density),dtype=np.int64)
    ind = np.unique(ind);
    j = ind // m
    i = ind - j * m
    vals = cp.ones(ind.size)
    return cupyx.scipy.sparse.coo_matrix(
        (vals, (cp.array(i.astype(np.int32)), cp.array(j.astype(np.int32)))),
        shape=(m, n), dtype=dtype).asformat('csr')
```

The second majour improvement is to optimize expresions of the form mask.multiply(L @ R). There is no sense computing the entire matrix product if we immediately throw out most of the entries and we do not have the RAM to store the result in the first place. To that end we will write a custom cuda kernel to perform the operation for each sparse coordinate in mask. This operation ends up computing the coefficients LR using a simple loop and should be improvable using more advanced CUDA libraries and/or memory management techniques. As a quick aside we must talk about how csr matrices are encoded.

$$\begin{aligned} \text{row number of } j^{th} \text{ entry } &= \max_i \text{csr\_matrix.indptr[i]} \leq j && (1) \\ \text{column index of } j^{th} \text{ entry } &= \text{csr\_matrix.indices[j]} && (2) \\ \text{value of } j^{th} \text{ entry } &= \text{csr\_matrix.data[j]} && (3) \end{aligned}$$

alternatively csr_matrix.indptr[i] $\leq j <$ csr_matrix.indptr[i+1] implies that the jth entry has column index i. As a further reminder, row and columns are 0 indexed in python and c++ while they are 1 indexed in mathematics.

```python
def csr_mult_dense_dense(sp: cupyx.scipy.sparse.csr_matrix, dnL: cp.ndarray, dnR: cp.ndarray)
                -> cupyx.scipy.sparse.csr_matrix:
    sp_m, sp_n = sp.shape
    dnL_m, dnL_n = dnL.shape
    dnR_m, dnR_n = dnR.shape
    # dnR_m, dnR_n = dnL_n, dnL_m
    # make sure that shapes match
    if sp_m != dnL_m or sp_n != dnR_n or dnL_n != dnR_m:
        raise RuntimeError('''dimension of sparse matrix must match dimenson or matrix product,
                    inner dimension of dense matrices must also match''')
    if sp.dtype != cp.float32 or dnR.dtype != cp.float32 or dnL.dtype != cp.float32:
        raise RuntimeError("data must be encoded as np.float32 or (float c type)")
    if dnL.flags.c_contiguous == False or dnR.flags.c_contiguous == False:
        raise RuntimeError("dense data must be c contiguous")
    nnz = sp.nnz
    dtype = np.promote_types(sp.dtype, np.promote_types(dnL.dtype, dnR.dtype));
    data = cp.zeros(nnz, dtype=dtype)
    indices = sp.indices.copy()
    indptr = sp.indptr.copy()
    # out = sp * dn
    kernel = cp.RawKernel(
            '''
            #include <cupy/complex.cuh>
            #include <cupy/carray.cuh>
            #include <cupy/atomics.cuh>
            #include <cupy/math_constants.h>

            __device__ inline int get_row_id(int i, int min, int max, const int *indptr) {
                int row = (min + max) / 2;
                while (min < max) {
                    if (i < indptr[row]) {
                        max = row - 1;
                    } else if (i >= indptr[row + 1]) {
                        min = row + 1;
                    } else {
                        break;
                    }
                    row = (min + max) / 2;
                }
                return row;
            }
            extern "C" __global__ void csr_mult_dense_dense(const float* SP_DATA,
            const int len,  const int* SP_INDPTR, const int* SP_INDICES,
            const int SP_M, const int SP_N,  const float* DN_DATA_L,
            const float* DN_DATA_R, const int DN_K, const int* OUT_INDPTR, float* OUT_DATA) {
                #pragma unroll 1
                CUPY_FOR(i, len) {
                    int m_out = get_row_id(i, 0, SP_M - 1, &(OUT_INDPTR[0]));
                    int n_out = SP_INDICES[i];
                    OUT_DATA[i] = 0;
                    for(int j = 0; j < DN_K; j++) {
                       OUT_DATA[i] += DN_DATA_L[m_out * DN_K + j] * DN_DATA_R[j * SP_N + n_out];
                    }
                    OUT_DATA[i] = OUT_DATA[i]*SP_DATA[i];
                };
            }''',
            'csr_mult_dense_dense', jitify=False)
    kernel((sp.data.size // 1024,),(1024,),(sp.data, sp.data.size, sp.indptr, sp.indices,
     sp_m, sp_n,  dnL.data, dnR.data, dnL_n, indptr, data))
    return cupyx.scipy.sparse.csr_matrix((data, indices, indptr), shape=(sp_m, sp_n))
```

CUPY_FOR is a macro that makes loops the thread index i by the number of threads over the length of the work array. cp.float32 is equivalent to float in C++, similairly cp.int32 maps to int. Writing custom kernels with CUPY is quite straightforward as long as you use the correct data types and the arrays are c-contiguous. Finally for testing purposes we need to measure the ℓ^2 distance between two pairs L_1R_1, L_2R_2. A quick way to optimize this operation is to compute the distances by row on separate threads and then sum them up.

```
def lr_dense_dense_diff_norm(dnL: cp.ndarray, dnR: cp.ndarray, dnL2:cp.ndarray, dnR2:cp.ndarray)
```

```
        -> cp.double:
    dnL_m, dnL_n = dnL.shape
    dnR_m, dnR_n = dnR.shape
    dnL2_m, dnL2_n = dnL2.shape
    dnR2_m, dnR2_n = dnR2.shape
    # dnR_m, dnR_n = dnL_n, dnL_m
    # make sure that shapes match
    if dnL_m != dnL2_m or dnL_n != dnL2_n:
        raise RuntimeError("dimensions of L should match L2")
    if dnR_m != dnR2_m or dnR_n != dnR2_n:
        raise RuntimeError("dimensions of R should match R2")
    if dnL_n != dnR_m:
        raise RuntimeError("inner dimensions must match")
    if dnL.dtype != cp.float32 or dnR.dtype != cp.float32 or dnL2.dtype != cp.float32
            or dnR2.dtype != cp.float32:
        raise RuntimeError("data must be encoded as np.float32 or (float c type)")
    dtype = cp.float32
    # allocate array to hold partial sums
    data = cp.empty(dnL_m, dtype=cp.double)
    # out = sp * dn
    kernel = cp.RawKernel(
            '''
            #include <cupy/complex.cuh>
            #include <cupy/carray.cuh>
            #include <cupy/atomics.cuh>
            #include <cupy/math_constants.h>

            extern "C" __global__ void lr_dense_dense_diff_partial(const double* DN_DATA_L,
                const double* DN_DATA_R,
                const double* DN_DATA_L2,
                const double* DN_DATA_R2,
                const int DN_M,
                const int DN_K,
                const int DN_N,
                double* VEC_OUT) {
                #pragma unroll 1
                CUPY_FOR(i, DN_M) {
                    VEC_OUT[i] = 0;
                    for(int k = 0; k < DN_N; k++) {
                        // compute difference of L@R[i, k] and L2@R2[i,k]
                        double val = 0;
                        for(int j = 0; j < DN_K; j++) {
                            val += DN_DATA_L[i * DN_K + j] * DN_DATA_R[j * DN_N + k];
                            val -= DN_DATA_L2[i * DN_K + j] * DN_DATA_R2[j * DN_N + k];
                        }
                        VEC_OUT[i] += val*val;
                    }
                };
            }''',
            'lr_dense_dense_diff_partial')
    kernel((dnL_m // 1024,),(1024,),(dnL.astype(cp.double).data, dnR.astype(cp.double).data,
                    dnL2.astype(cp.double).data, dnR2.astype(cp.double).data, dnL_m, dnL_n, dnR_n, data))
    return data.sum() ** 0.5;
```

The new itteration code is very similar to before, we just changed some portions to their rawkernel equivalents

```
# X: reduced rank left hand operand M*R matrix from previous step or random guess
# Y: reduced rank right hand operand R*N matrix from previous step or random guess
# mask: sparse {0,1} valued matrix aligned with sparse target
# Z: M*N sparse matrix with values aligned with mask.
def ASD_iteration(X: cp.ndarray, Y: cp.ndarray, mask: cp.sparse, Z_0: cp.sparse):
    # gradients of steppest ascent for objective function with that argument held fixed
    grad_Y_f = (csr_mult_dense_dense(mask, X, Y) - Z_0) @ (Y.T) # this result is not c-contiguous
    t_x = cp.linalg.norm(grad_Y_f) ** 2 / linalg.norm(csr_mult_dense_dense(mask,
            cp.ascontiguousarray(grad_Y_f), Y)) ** 2
    X_1  = X - t_x * grad_Y_f
    grad_X_f = X_1.T @ (csr_mult_dense_dense(mask, X_1, Y) - Z_0) # this result is not c-contiguous
    t_y = cp.linalg.norm(grad_X_f) ** 2 / linalg.norm(csr_mult_dense_dense(mask, X_1,
             cp.ascontiguousarray(grad_X_f))) ** 2
    Y_1 = Y - t_y * grad_X_f
    return X_1, Y_1

#
def random_test(M: int, N: int, R: int, D: float, MAX_ITERATIONS: int):
    # generate a random matrix with specified max rank
    # true rank of the matrix may end up being lower but it is unlikely
    # however distribution of singular values will probably not be very uniform
    truthL,truthR = createRandomLRMaxRank(M, N, R)
    # generate a random sparse mask to act as a random sample of our low rank matrix
    mask = createRandomSparseMask(M, N, D)
    # perform a Hadamard (componentwise) multiplication to get a sparse sampling of our low rank matrix
    observed = csr_mult_dense_dense(mask, truthL, truthR)

    # create a random guess for X_0 and Y_0 as our starting point
    # in practise these should be scaled to be comparable to the data
    X_0,Y_0 = createRandomLRMaxRank(M, N, R);
    errors = np.zeros(MAX_ITERATIONS)
    errors2 = np.zeros(MAX_ITERATIONS)
    #benchmarking
    startTime = datetime.now()
    for x in range(MAX_ITERATIONS):
        # errors[x] = cp.linalg.norm((cp.matmul(X_0, Y_0)) - truth)
        errors2[x] = linalg.norm(csr_mult_dense_dense(mask, X_0, Y_0) - observed)
        # cost is too high to compute per cycle for large data
        # errors[x] = lr_dense_dense_diff_norm(X_0, Y_0, truthL, truthR)
        X_0, Y_0 = ASD_iteration(X_0, Y_0, mask, observed)

    # compute error from truth, this operation is expensive and should not be performed each iteration
    print("total error: ")
    print(lr_dense_dense_diff_norm(X_0, Y_0, truthL, truthR))
    # plot the error from observed
    plt.plot(errors2, label ="error from observed (norm)", color= (1,0,0), linewidth=0.5)
    plt.yscale("log")
    plt.xlabel("iterations", fontsize='8')
    plt.legend(fontsize='10')
    plt.title("Rank " + str(R) +" matrix completion for " + str(M) + " by "
          + str(N) + " with " + str(D*100) + "% known values", fontsize='10')
    print("completed in: ")
    print(datetime.now() - startTime)
    plt.show()
```