cuRSeurat

Simone Avesani

Introduction

GPU parallelization has opened a new frontier in Computer Science. The possibility to speed up the execution of complex operations working in parallel has allowed to develop more efficient programming libraries and software. Today CUDA language is used in a lot of different research fields including Bioinformatics and for this reason is increasingly important the development of new libraries which provide the inter-operability between CUDA and other programming languages. In this context is born *cuRSeurat* a simple R package which provides the CUDA implementation of some functions of *Seurat*, one of the most used R package in the field of Single Cell analysis.

Package structure

Package *cuRSeurat* provides six different CUDA functions which could be executed independently. All the functions, developed with CUDA language with the aim of reduce the execution time, are called by six different C++ functions which were then integrated with R language using the R package *Rcpp*.
This package provides a set of constructs useful for the mapping between R and C++ data types using the R interface *.Call*.
For the development of *cuRSeurat* package was used a specific extension of *Rcpp*, a package called *RcppEigen*, which provides all the functionalities of *Rcpp* with the addition of some functions useful for the mapping between R and C++ objects of the library *Eigen*, very convenient working with sparse and dense matrices.

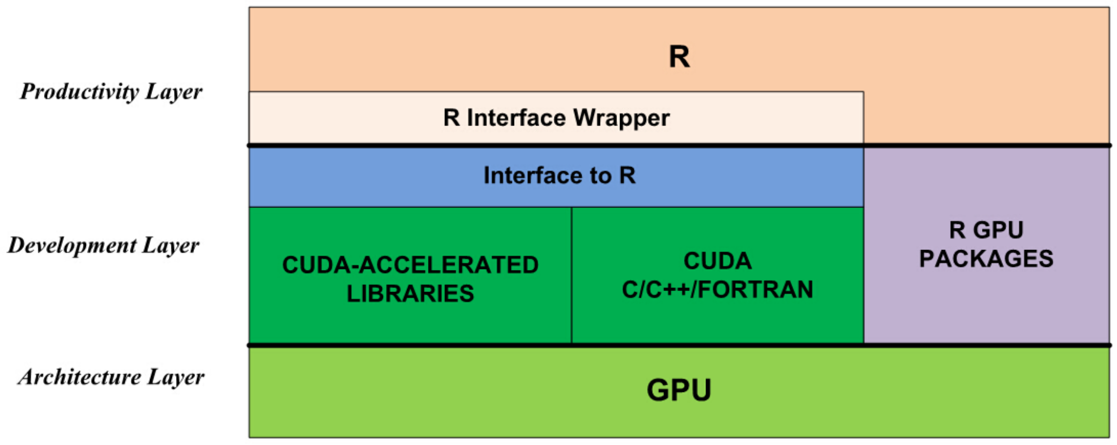

Figure 1: R-Cuda interaction

Implementation

All the functions of the package involve matrix operations, so it was decided to use the C++ library *Eigen* which provides a lot of constructs and data structure useful to manipulate and work with sparse and dense matrices. The six functions developed are:

- **LogNorm_cu**: performs the logarithmic normalization of a sparse matrix;
- **FastSparseRowScale_cu**: performs the row scaling of a sparse matrix;
- **FastRowScale_cu**: performs the row scaling of a dense matrix;
- **FastExpMean_cu**: calculates the mean expression of each gene inside a sparse matrix where genes are represented by the rows;
- **SparseRowVar2sd_cu**: calculates the gene variance inside a sparse matrix using the standard deviation;
- **SNN_cu**: creates a *Shared Nearest Neighbor* matrix starting from a sparse matrix;

For each of these functions different implementation choices were taken and different difficulties encountered. In the next sections we analyze each function in detail.

SparseMatrix

A sparse matrix is a large matrix for which are stored, to reduce memory consumption, only values different from zero. *Eigen* library provides a specific data structure useful to manipulate this specific type of matrices. Each *Eigen* sparse matrix is stored in memory as four compact arrays:

- **Values**: stores the coefficient values of non-zeros;
- **InnerIndices**: stores the row index of each non-zero value;
- **OuterStarts**: stores for each column the index of the first non-zero in the *Values array*;
- **InnerNNZs**: stores the number of non-zeros of each column;

This specific storage format is called CSC and the "_" that we find inside the values array indicates available free space to quickly insert new elements. The case where no empty space is available is a special case, and is refered as the compressed mode. It corresponds to the widely used Compressed Column (or Row) Storage schemes (CCS or CRS). Any sparse matrix can be turned to this form by calling the compression function. In the case in which we work with this specific compressed object type, *InnerNNZs* array is redundant with *OuterStarts* because we have the equality:

$$InnerNNZs[j] = OuterStarts[j+1] - OuterStarts[j] \tag{1}$$

Moreover, each operation between sparse matrices produce a compressed object, which is characterized by a *Values* vector and a *InnerIndices* vector of lengths number of non-zero values, while the *OuterStarts* vector and the *InnerNNZs* vector are respectively of sizes number of columns + 1 and number of columns. *Eigen* library provides a lot of different functions useful to make operations between sparse matrices but as it is possible to create column-major (default) or row-major objects, the majority of arithmetic operations on sparse matrices will assert that they have the same storage order. The library also provides three useful functions for the interoperability between classes and also useful to work with Eigen's sparse matrices in CUDA environments:

- **valuePtr()**: returns a pointer to the *Values* array;
- **innerIndextr()**: return a pointer to the *InnerIndices* array;
- **outerIndexPtr()**: return a pointer to the *OuterStarts* array;

Note that if the matrix is not in compressed form, compression function should be called before.

LogNorm_cu

The function *LogNorm_cu* performs the logarithmic normalization of a sparse matrix.

```
Eigen::SparseMatrix<double> LogNorm(Eigen::SparseMatrix<double> data, int scale_factor){
  Eigen::VectorXd colSums = data.transpose() * Eigen::VectorXd::Ones(data.rows());
  for (int k=0; k < data.outerSize(); ++k){
    for (Eigen::SparseMatrix<double>::InnerIterator it(data, k); it; ++it){
      it.valueRef() = log1p(double(it.value()) / colSums[k] * scale_factor);
    }
  }
  return data;
 }
```

As we can see the sequential code could be divided in two main parts. The first part computes an array of length equal to the number of columns of the matrix in which are stored the sums of the values of each column.

```
Eigen::VectorXd colSums = data.transpose() * Eigen::VectorXd::Ones(data.rows());
```

To compute the columns sums array the simple idea was to transpose the matrix and multiply it for a vector of length number of matrix columns, filled with ones. In this way is possible to compute an array of size number of matrix columns filled with the values obtained multiplying each row of the matrix (columns before transposition) by the vector of ones. To parallelize this specific step, has been created a specific kernel which takes in input the pointer to *Values* array, the pointer to *OuterStarts* array and the number of columns of the matrix. Each thread computes the sum of each column and at the end it store the sum into a specific array.

```
__global__ void cu_colSum(double *input, int *col , int *sum,  int ncol){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for(int i=idx; i<ncol; i += blockDim.x * gridDim.x){
        sum[i] = 0;
        for(int j= col[idx]; j<col[idx+1]; j++){
            sum[i] += input[j];
        }
    }
}
```

To know the number of non-zero values for each column, is used a simple for loop in which, for each thread, we scan the *Values* array for a number of positions equals to the difference between the value in the

column position and its successor inside the *OuterStarts* vector. To improve the portability and the capability of the function to work with huge matrices a grid-stride loop is added to the function.
The second part of the sequential function scan each value of the matrix and normalize it dividing the value by the product between the sum of the corresponding column and a scale factor. At the end the logarithm of the value obtained is computed and the matrix is returned. To parallelize this second part was created another kernel in which each thread computes the normalization of one values of the matrix.

```
__global__ void cu_fun(double *input,
                        double *out,
                        long int N,
                        int *colSum,
                        int *pos,
                        int scale_factor){

    int idx =  blockIdx.x * blockDim.x + threadIdx.x;
    for(int i = idx; i<N; i+=blockDim.x * gridDim.x ){
        out[i] = log1p(double(input[i]) / colSum[pos[i]] * scale_factor);

    }
}
```

The critical step of this function is to know in which column is a specific non-zero value inside the sparse matrix. To obtain this specific information another vector of length equal to the number of non-zero values was computed outside the kernel in which we can find, in each position, the corresponding column index of each non-zero value. This vector was computed between the two kernels with the aim to reduce at minimum the overhead produced. As before a grid-stride loop is used to improve the portability of the kernel.

FastSparseRowScale_cu

The second CUDA function implemented computes the row scaling of a sparse matrix. As for the normalization, we could divide the analysis of the function in two different parts.
In the first step we measure for each row of the sparse matrix the mean and the standard deviation taking into account if we want to scale and center the matrix. For the CUDA implementation, to compute this specific part, a single kernel was developed . A very interesting aspect that we need to observe before analyzing the kernel is that at the beginning and at the end of the function the matrix must be transposed, this is a very crucial operation because we can not compute the row mean or standard deviation of a column-major sparse matrix, the only way to do it is to transpose the matrix and work on its columns.

```
_global__ void cu_scale_sparse_factors(double *input, int *col ,
                                       int ncol,
                                       int nrow,
                                       int N,
                                       bool scale,
                                       bool center,
                                       double *colMean,
                                       double *colSdev){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for(int k=idx; k<ncol; k += blockDim.x * gridDim.x ){
      colMean[k] = 0;
```

```
        colSdev[k] = 0;
        for(int i= col[k]; i<col[k+1]; i++){
            colMean[k] += input[i];
        }
        colMean[k] = colMean[k]/nrow;

        if(scale == true){
            int nnZero = 0;
            if(center == true){
                for(int i=col[k]; i<col[k+1]; i++){
                    nnZero +=1;
                    colSdev[k] += pow((input[i] - colMean[k]), 2);
                }
                colSdev[k] += pow(colMean[k], 2) * (nrow - nnZero);
            }
            else{
                for(int i=col[k]; i<col[k+1]; i++){
                    colSdev[k] += pow(input[i], 2);
                }
            }
            colSdev[k] = sqrt(colSdev[k] / (nrow - 1));
        }
        else{
            colSdev[k] = 1;
        }
        if(center == false){
            colMean[k] = 0;
        }
      }
   }
```

As we can see from the kernel, each thread computes the mean and standard deviation of one column of the transposed matrix, the pointer *input* is a pointer to the *Values* array and pointer *col* point to the *OuterStarts* array. Mean and standard deviation values where then stored inside two specific vectors of length equal to the number of columns (rows of the original matrix) of the matrix. Depending on whether we want to scale and center the matrix, mean and standard deviation values are updated. As for all the other kernels a grid-stride loop is used for a better portability of the code.
During the second step of the scaling function e the two measures computed before are used to scale each value of the sparse matrix and at the end the function checks if the scaled values are greater then a limit value.

```
    void cu_scaling(double *input,
                    int *pos,
                    double *colMean,
                    double *colSdev,
                    int N,
                    int max ) {

  int idx = blockIdx.x * blockDim.x + threadIdx.x; //col

    for(int k=idx; k<N; k += blockDim.x * gridDim.x){

        input[k] = (input[k]- colMean[pos[k]]) / colSdev[pos[k]];
        if(input[k] > max){
            input[k] = max;
        }
    }

}
```

The kernel takes in input four different pointers: *input* points to the *Values* array, *colMean* points to the vector of columns means, *colSdev* points to the vector of standard deviation values and *pos* points to a vector of length equal to the number of non-zeros values that contain the index of the column of each value, as made for the normalization function the position vector is computed between the two kernel with the aim to reduce the overhead. The two remaining inputs are the number of non-zero values and the bound scaling value. Inside the kernel is possible to observe a grid-stride loop which introduce to the scaling operation. Each thread takes one value of the matrix and scales it making the difference between the original value and the row mean and dividing the result by the standard deviation of the row. The last step of the kernel is to check if each value is lower than a specific bound, if greater it is updated with the bound value.

FastRowScale_cu

The function *FastRowScale_cu* compute a dense scaled matrix starting from a dense matrix. The main steps of the functions are the more or less the same of the *FastSparseRowScale_cu* function but in this case is not necessary to transpose the matrix, working with a dense matrix is possible to compute row means and standard deviation without any problem. As before two different kernels were computed, the first creates two arrays of size equal to the number of rows of the matrix and fill them with the mean and standard deviation of each row.

```
__global__ void cu_scale_noSparse_factors(double *input,
                                          int nrow,
                                          int ncol,
                                          bool scale,
                                          bool center,
                                          double *mean,
                                          double *std){

    int idy = blockIdx.y * blockDim.y + threadIdx.y; //row

    for(int k=idy; k<nrow; k +=  blockDim.y * gridDim.y){
        mean[k] = 0;
        for(int i= 0; i < ncol; i++){
            mean[k+ += input[(i*nrow)+k];
        }
        mean[k] = mean[k]/ncol;
        if(scale == true){
            if(center == true){
                for(int i=0; i<ncol; i++){
                    std[k] += pow((input[(i*nrow)+k]-mean[k]),2);
                }
                std[k] = sqrt(std[k]/(ncol - 1));
            }
            else{
                for(int i=0; i< ncol; i++){
                    std[k] += pow(input[(i*nrow)+k], 2);
                }
                std[k] = sqrt(std[k]/(ncol - 1));
            }
        }

        if(center == false){
            mean[k] = 0;
        }
    }
}
```

As kernel before, this takes in input a pointer to the *Values* array and each thread computes the mean and standard deviation for each row of the matrix. The values computed changes depending on if we want to scale and center the matrix and at the end of the execution the function has computed two arrays containing the mean and standard deviation of each row. The second step of the scaling process is to scale the function using the values just calculated.

```
__global__
void cu_scaling_noSparse(int max,
                         double *m,
                         int nrow,
                         int ncol,
                         double *mean,
                         double *std) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x; //col
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    for(int k =idx; k<ncol; k +=  blockDim.x * gridDim.x){
        for(int j=idy; j<nrow; j+=  blockDim.y * gridDim.y){
            int x  = k*nrow + j;
            m[x] = (m[x]- mean[j])/ std[j];

            if(m[x] > max){
                m[x] = max;
            }
        }
    }

}
```

Like for the sparse matrices each thread scales one value of the matrix making the difference between the value and the row mean and dividing the result obtained by the corresponding standard deviation. At the end, each value is compared with a scaling bound and updated to it if greater. In this case, working with a dense matrix, there are no problems for the identification of the row index of each value so it's not necessary to compute a position vector in which storing the index of the column (row in matrix before transposition) of each value of the matrix.

FastExpMean_cu

The function *FastExpMean_cu* computes the expression mean of each gene that is equivalent to the mean of each row inside a sparse matrix.

```
Eigen::VectorXd FastExpMean(Eigen::SparseMatrix<double> mat){
  int ncols = mat.cols();
  Eigen::VectorXd rowmeans(mat.rows());
  mat = mat.transpose();
  for (int k=0; k<mat.outerSize(); ++k){
    double rm = 0;
    for (Eigen::SparseMatrix<double>::InnerIterator it(mat,k); it; ++it){
      rm += expm1(it.value());
    }
    rm = rm / ncols;
    rowmeans[k] = log1p(rm);
  }
  return(rowmeans);
}
```

```
__global__
void cu_fastmean(double *input , int *col, double *mean, int ncol, int nrow  ){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    double colSum;
    double nnZero;

    for(int k = idx; k<ncol; k += blockDim.x * gridDim.x){
        for(int i = col[k]; i < col[k+1]; i++){
                mean[k] += expm1(input[i]);
        }

        mean[k] = mean[k]/nrow;
        mean[k] = log1p(mean[k]);

    }
}
```

As we can see inside the kernel each thread computes the mean of each row, as we are working with a sparse matrix before to compute the mean we must transpose the matrix in a way to work with the columns of the matrix. The mean computed is not the standard mean, before to sum each value inside a column we calculates its exponential and at the end we make the logarithm of each row mean. As for the other functions a grid-stride loop is used.

SparseRowVar2sd_cu

The fifth function implemented called *SparseRowVar2sd_cu* takes, as input, a sparse matrix and computes the gene variance that is the variance inside each row of the matrix. This specific function has been implemented using a unique kernel that computes an array of length equal to the number of rows of the matrix filled with the corresponding row variance.

```
NumericVector SparseRowVar2(Eigen::SparseMatrix<double> mat,
                            NumericVector mu){

  mat = mat.transpose();
  NumericVector allVars = no_init(mat.cols());
  for (int k=0; k<mat.outerSize(); ++k){
    double colSum = 0;
    int nZero = mat.rows();
    for (Eigen::SparseMatrix<double>::InnerIterator it(mat,k); it; ++it) {
      nZero -= 1;
      colSum += pow(it.value() - mu[k], 2);
    }
    colSum += pow(mu[k], 2) * nZero;
    allVars[k] = colSum / (mat.rows() - 1);
  }

  return(allVars);
}
```

```
__global__
void cu_sparseVar_sd(double *input,
                     int *col,
                     double *mu,
                     int nrow,
                     int N,
                     double * v,
                     double *sd,
                     int vmax){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    double colSum;
    double nnZero;

    for(int k = idx; k<N; k += blockDim.x * gridDim.x){
        if(sd[k] != 0){
            double colSum = 0;
            int nnZero = nrow;
            for(int i = col[k]; i < col[k+1]; i++){
                nnZero -= 1;
                if(vmax < (input[i] - mu[k]) / sd[k] ){
                    colSum += pow(vmax, 2);
                }
                else{
                    colSum += pow( (input[i] - mu[k]) / sd[k], 2);
                }
            }
            colSum += pow((0 - mu[k]) / sd[k], 2) * nnZero;
            v[k] = colSum / (nrow -1);
        }


    }

}
```

The kernel takes in input, in addition to the pointer pointing to the *Values* array, two other pointers *mu* and *sd* which point respectively to a row means vector and to a row standard deviations vector. This two arrays have already been pre-computed by another function and are used to carry out the variance of each matrix row. As before, since we are working with a sparse matrix, we have to transpose the matrix before and after the execution of the kernel.

SNN_cu

The last function implemented is called *SNN_cu*, it computes the *Shared Nearest Neighbor* matrix starting from a sparse matrix. The *Shared Nearest Neighbor* matrix will be used later to compute the clustering algorithm Shared nearest neighbors.

```
Eigen::SparseMatrix<double> ComputeSNN(Eigen::SparseMatrix<double> SNN, double prune) {
  int k = SNN.cols();
  SNN = SNN * (SNN.transpose());
  for (int i=0; i < SNN.outerSize(); ++i){
    for (Eigen::SparseMatrix<double>::InnerIterator it(SNN, i); it; ++it){
      it.valueRef() = it.value()/(k + (k - it.value()));
      if(it.value() < prune){
        it.valueRef() = 0;
      }
    }
  }
  SNN.prune(0.0); // actually remove pruned values
  return SNN;
}
```

As we can observe from the sequential implementation of the function, two different steps are required.

In the first step a sparse is generated multiplying the input sparse matrix by its transpose.

```
  SNN = SNN * (SNN.transpose());
```

The second step is the preprocessing of the matrix obtained during the step one. During this step each value of the new matrix is divided by the sum between the number k (the number of columns of the original matrix) and the difference between k and value itself. To compute this function two different kernels were implemented, the first kernel computes the product between one sparse matrix and its transposition, while the second computes the preprocessing step.

```
__global__
void cu_multSparse(double *A , int *col, double *out, int ncol ){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    double sum;

    for(int j= idx; j<ncol; j += blockDim.x * gridDim.x){
        for(int k = idy; k<ncol; k += blockDim.y * gridDim.y){
            int row_start = col[j];
            int row_end = col[j+1];
            sum = 0;

            for(int i = row_start; i < row_end; i++){
                sum += A[i] * A[col[k] +(i-row_start)];
            }

            out[j*ncol +k] = sum;
        }
    }
}
```

The kernel above computes the sparse matrix product, it takes in input the pointer *A* pointing to the *Values* array, the pointer *col* pointing to the *OuterStarts* array, the number of columns of the transposed matrix and a pointer to the matrix obtained by the product. To make this specific product we can not multiply each row by each column but the only possibility is to multiply each column by each other column, for this reason the input matrix is only one and it is the transposed matrix of the original sparse matrix. Each thread works on a column, so in total 2* number of columns threads are used and each column is multiplied by all the other columns and by itself. The final result is a square matrix obtained from the product of each columns combination.
The second kernel computes the preprocessing step of the matrix.

```
__global__
void cu_nSparse(double *in, int *col, int k, int N, int prune, double *out){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for(int j= idx; j<N; j += blockDim.x * gridDim.x){
        for(int i=col[j]; i<col[j+1]; i++){
            out[i] = in[i]/(k + (k - in[i]));
            if(out[i] < prune){
                out[i] = 0;
            }
        }
    }
}
```

It takes in input the matrix computed in the previous kernel, a pointer *col* pointing to the *OuterStarts* array, the integer k that is number of columns of the original sparse matrix and some other parameters useful for the computation. Each thread works on one value of the matrix and process it. As for all the other kernels that work on a sparse matrix, the pointer *col* is essential to know which are the values inside a specific column.

Results

In this section I analyze the results obtained in terms of speedup for each function. But before, some considerations need to be made, in particular we need to consider that in each operation which involves a sparse matrix, it has been necessary to compressed it to ensure the interoperability between the object and CUDA. This step, not necessary in the sequential code, introduce a significant overhead that, in some cases, increases the overall execution time (it required more or less 0.030 sec).
Another important aspect that we need to consider is the delay measured during the first execution of a kernel. During the testing phase of the R package functions I observed a huge and unexpected delay during the execution of the first R function, this is due to the fact that at the first execution of a kernel inside a R session, all necessary CUDA libraries are loaded significantly increasing the overall time about more or less 70 ms. To avoid this problem was included into the package a ”warm up” function which runs a very simple kernel used for the libraries loading process. To obtain better performances from the package functions is recommended to execute the ”warm up” function at the beginning of each R session.
All the functions were tested on a matrix with 2700 columns and 13000 rows provides by the Seurat package using a GTX 950m GPU model.

LogNorm_cu

	C++/CUDA function	**R function**
Sequential	0.231 sec	0.442 sec
Parallel	0.045 sec	0.213 sec
Speedup		2.07x

Normalization function works on a sparse matrix so we have to take into account the time required for the compression and also the time required to compute the index of the column for each non-zero.

FastSparseRowScale_cu

	C++/CUDA function	**R function**
Sequential	7.842 sec	7.980 sec
Parallel	0.432 sec	0.593 sec
Speedup		13.45x

FastRowScale_cu

	C++/CUDA function	**R function**
Sequential	9.990 sec	10.479 sec
Parallel	0.793 sec	1.444 sec
Speedup		7.25x

FastExpMean_cu

	C++/CUDA function	**R function**
Sequential	0.099 sec	0.232 sec
Parallel	0.071 sec	0.208 sec
Speedup		1.12x

As we can observe from the table, *FastExpMean_cu* function do not exhibit a significant speedup, this is probably due to the overhead caused by compression that in this simple function is more evident. A more efficient parallelization can surely be obtained working on the transposition of the sparse matrix.

SparseRowVar2sd_cu

	C++/CUDA function	**R function**
Sequential	0.190 sec	0.291 sec
Parallel	0.151 sec	0.263 sec
Speedup		1.11x

As for the previous function the speedup is not significant, as before I think that this is due to the overhead generated by the compression respect to the simple structure of the function. As before a good improvement could be parallelize the transposition process of a sparse matrix.

SNN_cu

	C++/CUDA function	**R function**
Sequential	28.423 sec	28.531 sec
Parallel	2.966 sec	3.221 sec
Speedup		8.85 x

To conclude we can say that the result for the majority of the function is good but I think that some improvements can be made for example implementing a CUDA kernel for sparse matrix transposition and also try to work with the shared memory in some kernel as for example in the function which computes the sparse matrices multiplication. Another interesting aspect could be try to understand if it is possible to work in CUDA with sparse matrices without compressed them with aim of remove some overhead inside the kernels.