Supporting Information File S1

Documented Source Code

In this section we will discuss the source code of the GPU-based fast and scalable kNN (FSkNN) method. We performed our computational tests on the following hardware setup. Four NVIDIA Tesla C2050 GPU cards installed on a Xenon Nitro T5 Super-micro server that has Dual Xeon 5620 2.4GHz processors, 32GB of 1066 MHz DDR3 RAM and 800GB of Local Hard Disk. The programs were written in C++ and CUDA and compiled using the `g++` 4.4.4 and `nvcc` compiler on a Linux OS, Kernel version 2.6.9.

Source code compilation. The code can be compiled using NVIDIA CUDA compiler driver `nvcc` release 3.0 and up. It will require OpenMP support ("`-fopenmp -lgomp`") to handle multiple GPUs and Boost library to parse the input files.

Algorithm: GPU-FSkNN

Algorithm 1: Fast and Scalable kNN Algorithm (FSkNN)

Input : `In`, input matrix or a portion of the input matrix;
$\mathtt{n}_{\mathtt{chunksize}}$, dimension of the sub-matrix (chunk);
`k`, number of nearest neighbors;

Output: The kNN graph stored in `Gk`;

```
1  create Gk;
2  Gk[i].weight ← float_max, for i← 0...(n_row − 1);
3  initialize (segments, n_GPU);
4  foreach segment ∈ segments do
5      create D;
6      create Gk′;
7      host → device (In, D, Gk′);
8      initialize (splits, segment);
9      create Maxk;
10     foreach split ∈ splits do
11         initialize Maxk;
12         host → device (Maxk);
13         initialize (chunks, split);
14         foreach chunk ∈ chunks do
15             Call Distance Kernel <<<grid1, block1>>> (In, n_chunksize,
               split, chunk, b);
16             Call kNN Kernel<<<grid2, block2>>>(D, n_chunksize,
               Maxk);
17     device → host (Gk′);
18 return Gk;
```

The algorithm starts with an input matrix `In` and produces a kNN graph (`Gk`). First, the `weight` attribute of each edge in the kNN graph is set to the maximum value of float (`float_max`) and the number of available GPUs ($\mathtt{n}_{\mathtt{GPU}}$) is assigned as the maximum number of segments (`segments`). The computational tasks for each of these segments are handled by separate GPUs. During the execution of each segment, the algorithm creates an array `D` (for holding a chunk of the matrix) and an additional pointer array `Gk′` linked to `Gk` (for holding the partial kNN graph of the respective segment) and transfers them to the device memory. Now to compute the partial kNN graph, the algorithm executes all the splits in each segment and subsequently all the chunks in each split. For each chunk the algorithm invokes the `Distance kernel` to compute a sub-matrix/chunk (`D`) and the `kNN Kernel` to compute an intermediate kNNs list. This list eventually becomes a partial kNN graph (stored in `Gk′`) when all the chunks in a split and all the splits in a segment are executed completely.Then, the `Gk′` is transferred to host and mapped to the respective location in `Gk`.

Implementation: fs$k$nn() function

```
1  /* fsknn function to represent the fsknn algorithm.
2   * Input prameters:
3   * @param filenameOut  : Output File name
4   * @param Ina          : Input matrix (nRow x nCol)
5   * @param nRow         : Total number of rows in the input matrix
6   * @param nCol         : Total number of columns in the input matrix
7   * @param chunkSize    : Total number of rows/columns in a chunk
8   * @param k            : The integer value of k.
9   * Output:
10  * Graph(kNN) will be writen in (filenameOut).knn file in
11  * the following format:
12  *
13  * number of vertices,  number of edges
14  * source, target, weight
15  * ....
16  * source, target, weight
17  */
18
19 int fsknn(char *filenameOut, float *Ina, int nRow, int nCol,
20           int chunkSize, int k)
21 {
22         /* define thread and blocks for Distance Kernel */
23         dim3 blocksD (chunkSize/b,chunkSize/b);
24         dim3 threadsD (b,b);
25
26         /* define thread and blocks for kNN Kernel */
27         int threadsK = 512; // maximum number of threads allowed
28         int blocksK = chunkSize / threadsK+1;
29
30         /* Declare local variables */
31         float3 *Gka;
32         int nRowExtended;
33         int nColExtended;
34         int nExtraCol;
35         int nExtraRow;
36         int splitSize;
37         int segmentSize;
38
39         /* Extend the columns */
40         nColExtended = nCol;
41         while (nColExtended % b !=0) {
42             nColExtended++;
43         }
44         nExtraCol = nColExtended - nCol;
45
46
47         /* Extend the rows */
```

```
48          nRowExtended = nRow;
49          while (nRowExtended % chunkSize !=0){
50              nRowExtended++;
51          }
52          nExtraRow = nRowExtended - nRow;
53
54          /* Identify total number of chunks in a split  */
55          splitSize = nRowExtended / chunkSize;
56
57          /* Allocate memory to store the kNN graph*/
58          Gka = (float3) malloc (sizeof(float3) * nRowExtended * k);
59
60          for (int i = 0; i < nRowExtended * k; i++)
61                  Gka[i].z = MaxValue; //initialize weights
62
63          /* Setup multi-GPUs  */
64          int nGpu = 0;
65          cudaGetDeviceCount(&nGpu); //get the number of GPUs
66
67          if(nGpu < 1){
68                  printf("no CUDA capable devices were detected!\n");
69                  return 0;
70          }
71
72          if(nGpu > splitSize) nGpu = splitSize; // for small matrices
73
74          /* Identify total number of splits in a segment  */
75          segmentSize = splitSize / nGpu;
76
77          /* show some statistics on host and device */
78          printf("number of host CPUs:\t%d\n", omp_get_num_procs());
79          printf("number of CUDA devices:\t%d\n", nGpu);
80
81          for(int i = 0; i < nGpu; i++)
82          {
83               cudaDeviceProp dprop;
84               cudaGetDeviceProperties(&dprop, i);
85               printf(" %d: %s\n", i, dprop.name);
86          }
87
88          printf("---------------------------\n");
89
90          printf("#chunks in each split (splitSize): %d\n",splitSize);
91          printf("#splits in each segment (segmentSize):%d\n",segmentSize);
92
93          printf("---------------------------\n");
94
95
96
97
```

```
98  /* Implementation of the FSkNN algorithm   */
99
100         /* Set the number of threads = the number of available GPUs */
101         omp_set_num_threads(nGpu);
102
103         /* outer loop, in parallel for each segment (GPU) */
104         #pragma omp parallel
105         {
106              /* Host variables */
107              float *Da;
108              int *Maxka;
109              float3 *Gka_sub; // a pointer  to the partial kNN graph
110              unsigned int Gka_sub_bytes;
111
112              /* Device variables    */
113              float *dev_Ina,  *dev_Da;
114              int *dev_maxArray;
115
116              int tid = omp_get_thread_num();
117              int numthread = omp_get_num_threads();
118              int gpuid = -1;
119
120              /* Set GPU by the thread id    */
121              cudaSetDevice(tid);
122              cudaGetDevice(&gpuid);
123
124              /* Allocate memory on host */
125              Da = (float*) malloc(sizeof(float)* chunkSize*chunkSize);
126              Maxka = (int*)malloc(sizeof(int)*chunkSize);
127
128              /* Allocate memory on device */
129              cudaMalloc((void **) &dev_Ina,nRow*nCol*sizeof(float));
130              cudaMalloc((void **) &dev_Da,
131                        chunkSize*chunkSize*sizeof(float));
132              cudaMalloc((void **) &dev_maxArray,
133                        chunkSize*sizeof(int));
134
135              /* Copy Ina[] and Da[] to  device */
136              cudaMemcpy(dev_Ina, Ina, nRow*nCol*sizeof(float),
137                        cudaMemcpyHostToDevice);
138              cudaMemcpy(dev_Da, Da,chunkSize*chunkSize*sizeof(float),
139                        cudaMemcpyHostToDevice);
140
141              int splitBegin = tid * segmentSize;
142              int splitEnd= splitBegin + segmentSize;
143
144
145
146
147
```

```
148             /*For last split*/
149             if (( splitSize - 1 )- splitEnd <= segmentSize )
150                 splitEnd = splitSize;
151
152             Gka_sub = Gka + (tid*chunkSize*k)*(splitSize/numthread);
153
154             /*  Calculate bytes used by the partial kNN graph */
155             Gka_sub_bytes = (((tid+1)* chunkSize * k
156                                 * (splitEnd-splitBegin))
157                                 - (tid*chunkSize * k
158                                 * (splitEnd - splitBegin)))
159                                 * sizeof( float3);
160             float3 * dev_Gka_sub = 0;
161             cudaMalloc(( void **) &dev_Gka_sub,Gka_sub_bytes );
162             cudaMemset (dev_Gka_sub,0,Gka_sub_bytes);
163             cudaMemcpy (dev_Gka_sub,Gka_sub,Gka_sub_bytes,
164                         cudaMemcpyHostToDevice);
165
166             /* Loop over all the splits in a segment*/
167             for (int split = splitBegin; split < splitEnd ; split++){
168
169                /* Copy Maxka[] to the device */
170                 for (int r=0;r<chunkSize;r++)
171                       *(Maxka + r) = (split - tid * segmentSize)
172                                       * k * chunkSize + r * k;
173                       cudaMemcpy (dev_maxArray, Maxka,
174                                   chunkSize * sizeof(int),
175                                   cudaMemcpyHostToDevice);
176
177                 /* Loop over all the chunks in a split*/
178                 for (int chunk = 0; chunk < splitSize; chunk ++){
179
180                       // Call Distance Kernel
181                       PearsonDistanceKernel <<<blocksD,threadsD>>>
182                       (dev_Ina,  dev_Da, chunkSize, split, chunk,
183                        nRow,nCol,nExtraCol, nColExtended);
184
185                       // Call kNN Kernel
186                       kNNKernel <<< blocksK, threadsK >>>
187                       (dev_Da,  dev_Gka_sub, dev_maxArray,
188                       chunkSize, nExtraRow, split, chunk, splitSize,
189                       nRow, tid,segmentSize, k);
190                 }
191             }
192            /*Copy back the partial kNN graph for a split*/
193            cudaMemcpy(Gka_sub, dev_Gka_sub,
194                       Gka_sub_bytes,cudaMemcpyDeviceToHost);
195
196
197
```

```
198 /* Deallocate device memory */
199             cudaFree(dev_Ina);
200             cudaFree(dev_Da);
201             cudaFree(dev_maxArray);
202             cudaFree(dev_Gka_sub);
203       }    // Close OpenMP loop
204
205
206 /*Write down the kNN graph into an external file (filename.knn)*/
207
208           FILE * pFile;
209           pFile = fopen (filenameOut,"w");
210
211           if (pFile!=NULL)  {
212                   fprintf(pFile,"%d %d\n",nRow, nRow * k);
213                   for (int i = 0; i < nRow*k; i++)
214
215                           // source, target and weight
216                           fprintf(pFile,"%d %d %f\n",
217                                   int(Gka[i].x),
218                                   int(Gka[i].y),
219                                   Gka[i].z);
220                   fclose (pFile);
221           }
222          return 0;
223 }
```

Distance Kernel Algorithm

Algorithm 2: `Distance kernel` Algorithm

Input : `In`, input matrix or a portion of the input matrix;
$\texttt{n}_{\texttt{chunksize}}$, dimension of the distance matrix chunk;
`split`, index of split;
`chunk`, index of chunk;
`b`, data block size;

Output: none, a chunk of the original distance matrix stored in `D`;

create two shared memory arrays, `X` and `Y` to load data blocks of size (`b` × `b`) from `In`;

/* Identify the thread and block indices */;

`bx` ← `blockIdx.x`, `by` ← `blockIdx.y`;

`tx` ← `threadIdx.x`, `ty` ← `threadIdx.y`;

/* Identify the data blocks to process */;

$\texttt{Xbegin} \leftarrow (\texttt{bx} + \texttt{n}_{\texttt{chunksize}}/\texttt{b} \times \texttt{chunk}) \times \texttt{b} \times \texttt{n}_{\texttt{Col}}$;

$\texttt{Ybegin} \leftarrow (\texttt{by} + \texttt{n}_{\texttt{chunksize}}/\texttt{b} \times \texttt{split}) \times \texttt{b} \times \texttt{n}_{\texttt{Col}}$;

$\texttt{Yend} \leftarrow \texttt{Yend} + \texttt{n}_{\texttt{col}} - 1$;

$\texttt{d} \leftarrow 0$;

for `y` ← `Ybegin` **by** `b`, `x` ← `Xbegin` ***to*** `Yend` **do**

  $\texttt{Y[ty][tx]} \leftarrow \texttt{In}[\texttt{y} + \texttt{ty} \times \texttt{n}_{\texttt{col}} + \texttt{tx}]$;

  $\texttt{X[tx][ty]} \leftarrow \texttt{In}[\texttt{x} + \texttt{ty} \times \texttt{n}_{\texttt{col}} + \texttt{tx}]$;

  `synchronize_threads()`;

  foreach *column* `i` ∈ $\texttt{n}_{\texttt{col}}$ **to** `b` **do**

    if *(*$\texttt{n}'_{\texttt{col}} - \texttt{n}_{\texttt{col}}$*)* ≠ 0 ***and*** `y` ≥ *(*`Ybegin` + $\texttt{n}'_{\texttt{col}} - \texttt{b}$*)* ***and*** `i` ≥ $(\texttt{n}_{\texttt{col}} - \texttt{n}'_{\texttt{col}} + \texttt{b})$ **then**

      continue `/* exclude extra (pad) columns */`;

    `d` ← `d` + *distance (or similarity) between data in* `Y[ty][i]` *and* `X[i][tx]`;

  `synchronize_threads()`;

/*Identify location of the computed distance in the chunk */;

$\texttt{index} \leftarrow \texttt{by} \times \texttt{b} \times \texttt{n}_{\texttt{chunksize}} + \texttt{ty} \times \texttt{b} + \texttt{bx} \times \texttt{b} + \texttt{tx}$;

$\texttt{D[index]} \leftarrow \texttt{d}$;

Computation of the Distance Kernel. The `Distance kernel` is presented as a template function which be adapted for several types distance measures, such as *Euclidean* or *Manhattan* distance or similarity measures that are based on *Pearson's* or *Spearman's* correlation. Here, each thread is responsible for computing a single distance in the matrix. The threads are organized in a 2-dimensional CUDA thread and block structure. Although the basic algorithm and the thread organization have been adapted from Chang et al. [1, 2], we modified it to compute a chunk of the distance matrix instead of the

complete distance matrix. The working procedure of the algorithm is simple, during each iteration, every thread block loads (b × b) sized data blocks from the input matrix In to single dimensional shared memory arrays X and Y. Then after synchronization and each thread starts to calculate and accumulate own partial distances in d . When the distance values are finalized, each thread stores d to the appropriate location in D. It should be noted here that the algorithms in [1, 2] were designed to work only with the data sets where the number of rows and columns are multiples of 16 only (i.e., the *data block* size, b=16). This limitation was imposed so that all threads in any half-warp (a *warp* = 32 threads) can access the data in a sequence. We modified the original algorithm by introducing padded input matrix rows and columns so that the algorithm can work with any number of rows and columns.

Implementation: Distance Kernel ()

```
1  /* Distance kernel (Implements Pearsons Correlation).
2   * This code has adapted from the paper "Compute pairwise Manhattan
3   * distance and Pearson correlation coefficient of data points with
4   * GPU" by Chang et al. (2009) but instead of computing the whole
5   * distance matrix, we compute a chunk of size (chunkSize x chunkSize)
6   * of the complete matrix.
7   *
8   * Input Parameters:
9   * @param Ina          : Input matrix (nRow x nCol)
10  * @param Da           : Distance matrix chunk (chunkSize x chunkSize)
11  * @param chunkSize    : Total number of rows in a chunk
12                          (must be multiple of b)
13  * @param split        : split ID
14  * @param chunk        : chunk ID
15  * @param nExtraCol    : No. of extra columns added afterwards
16  * @param nColExtended : Total number of columns in the
17                          extended input matrix
18  * Output:
19  * none, Da[] left on the device memory
20  */
21
22
23
24 // Computes Pearsons Correlation
25 __global__ void PearsonDistanceKernel(float *Ina, float *Da,
26                 int chunkSize, int split, int chunk, int nRow,
27                 int nCol, int nExtraCol, int nColExtended)
28 {
29
30   __shared__ float Xs[b][b];
31   __shared__ float Ys[b][b];
32
```

```
33    int bx = blockIdx.x, by = blockIdx.y;
34    int tx = threadIdx.x, ty = threadIdx.y;
35    /* Adapted from Chang et al. (2009)  */
36    int xBegin = (bx + (chunkSize/b) * chunk) * b * nCol;
37    int yBegin = (by + (chunkSize/b) * split) * b * nCol;
38
39    int yEnd = yBegin + nCol - 1;
40    int x, y, i, index;
41
42    float a1, a2, a3, a4, a5;
43    float avgX, avgY, varX, varY, cov, rho;
44
45    a1 = a2 = a3 = a4 = a5 = 0.0;
46    for(y=yBegin,x=xBegin;y<=yEnd;y+=b,x+=b){
47
48          /* load partial data into shared memory */
49           Ys[ty][tx] = Ina[y + ty*nCol + tx];
50           Xs[tx][ty] = Ina[x + ty*nCol + tx];
51           __syncthreads();
52
53           /* Calculate and accumulate partial similarity/distance */
54           for(i = 0; i < b; i++ ){
55              if (nExtraCol!=0
56                 && (y>=(yBegin +(nColExtended-b)))
57                 && (i>= nCol-(nColExtended-b)))
58                    continue;
59
60              a1 += Xs[i][tx];
61              a2 += Ys[ty][i];
62              a3 += Xs[i][tx] * Xs[i][tx];
63              a4 += Ys[ty][i] * Ys[ty][i];
64              a5 += Xs[i][tx] * Ys[ty][i];
65          }
66      __syncthreads();
67
68    }
69    avgX = a1/nCol;
70    avgY = a2/nCol;
71    varX = (a3-avgX*avgX*nCol)/(nCol-1);
72    varY = (a4-avgY*avgY*nCol)/(nCol-1);
73    cov  = (a5-avgX*avgY*nCol)/(nCol-1);
74    rho  = cov/sqrtf(varX*varY);
75
76    /* Identify the location (index) of the computed distance
77    in the chunk of distance matrix */
78
79    index = by*b*chunkSize + ty*chunkSize + bx*b + tx;
80    Da[index] = rho;
81
82  }
```

*k*NN Kernel Algorithm

Algorithm 3: *k*`NN kernel` Algorithm

Input : `D`, a chunk of the original distance matrix;
$\texttt{n}_{\texttt{chunksize}}$, dimension of the chunk;
`Maxk`, an array to hold the farthest neighbors for each row index in the chunks;

Output: none, an (intermediate) kNN graph stored in $\texttt{Gk}'$;

```
1  row' ← blockIdx.x × blockDim.x + threadIdx.x;
2  if row' < n_chunksize then
3      initialize row /* absolute index in the original distance
       matrix */;
4      for column' ← 1 to n_chunksize do
5          initialize column /* absolute index in the original
           distance matrix */;
6          if row ← column or row > n_row or column > n_col then
7              continue /* exclude diagonal and pad regions */;
8          if D[row' × n_chunksize + column'] < Gk'[Maxk[row']].weight then
9              Gk'[Maxk[row']].source ← row;
10             Gk'[Maxk[row']].target ← column;
11             Gk'[Maxk[row']].weight ← D[row' × n_chunksize + column'];
12             Search the new maximum element in row'(D) and store the
               index in Maxk[row'];
```

Computation of the kNNs from a distance matrix chunk: The `kNN Kernel` algorithm utilizes an 1-dimensional thread and block structure. Here, each thread works on a single row (chunk) and identifies the k-nearest neighbors for respective row index, where an array `Maxk` holds the location of the farthest k-neighbor. For each row index (chunk) the respective farthest k-neighbor is investigated and replaced if the distance to any element$[i]$, $i \leftarrow 1 \ldots n_{chunksize}$ is found smaller. However, every index is not checked, rather based on the position of chunk in the original distance matrix the algorithm skips certain indices, for example, it excludes the diagonal indices (i.e., the distance from the point itself) for the chunks in diagonal positions and similarly it exclude the indices of the extra (pad) regions in the original distance matrix.

Implementation: kNN Kernel

```
/*
 *
 * Implements kNNKernel Algorithm
 *
 * Input Parameters:
 *
 * @param Da           : Distance matrix chunk (chunkSize x chunkSize)
 * @param Gka          : An array of 3-tuples {source,target,weight}
                         to store the kNN graph (Gka)
 * @param Maxka        : An array that contains the index of the
                         farthest nearest neighbours of
                         each node (in a chunk)
 * @param chunkSize    : Total number of rows in a chunk
 * @param nRow         : Total rows in the original distance matrix
 * @param nExtraRow    : Extra rows added to fit all the chunks
 * @param split        : split ID
 * @param chunk        : chunk ID
 * @param splitSize    : Total number of chunks in a split
 * @param chunkSize    : Total number of rows in a chunk
 * @param tid          : Openmp threadID (segment/gpu ID)
 * @param k            : Integer value of k
 *
 * Ouput:
 * none, kNN graph (Gka) will be left on the device.
 *
 *
 */

__global__ void kNNKernel (float* Da,  float3* Gka, int *Maxka,
                           int chunkSize, int nExtraRow, int split,
                           int chunk, int splitSize,  int nRow,
                           int tid, int segmentSize, int k)
{

/* Identify the row by thread id*/

int row = blockIdx.x*blockDim.x + threadIdx.x;

/* If the identified row belongs to the chunk*/
if (row < chunkSize)
   {

      /* Set the search range */
      int beginSearch = (split - tid * segmentSize) * k
                        * chunkSize + row * k ;
      int endSearch   = beginSearch + k;
      /*Search by Columns of the chunk*/
```

```
48        for(int column=0;column < chunkSize ; column++){
49
50            /* For Chunks in the diagonal of the original matrix */
51            if(split == chunk){
52
53               if(row != column){ //skip distance from the point itself
54
55                   /* Exclude the diagonal and extra regions
56                   of the matrix*/
57
58                   if (( nExtraRow !=0 )
59                         && ((split >= splitSize - 1 )
60                            || (chunk >= splitSize - 1 ))
61                               && (row >= chunkSize - nExtraRow
62                                  || column >= chunkSize-nExtraRow ))
63                                     continue;
64
65
66
67                   if(( *(Gka + *(Maxka + row ))).z >
68                            *(Da + chunkSize * row + column )) {
69
70                                ( *(Gka + *(Maxka + row))).z
71                                   = *(Da + chunkSize * row + column );
72
73                                ( *(Gka + *(Maxka + row))).x
74                                   = split * chunkSize + row;
75
76                                ( *(Gka + *(Maxka + row))).y
77                                   = chunk * chunkSize + column;
78
79                                /* Search the index of the max element
80                                within range */
81                                for (int index = beginSearch;
82                                         index < endSearch;
83                                         index++){
84
85                                         if(( *(Gka+index)).z >
86                                         (*(Gka + *(Maxka + row))).z )
87                                            *(Maxka+row) = index;
88
89                                         } // inner for loop
90
91                        } //if
92
93                 }  // if row != column
94
95            } // if split == chunk
96            /* For chunks in non-diagonal positions of
97               the original matrix */
```

```
98
99            else
100           {
101                /* Exclude the extra regions of the matrix*/
102
103               if ((nExtraRow != 0)
104               && (column >= chunkSize - nExtraRow
105                 &&  chunk >= splitSize-1)
106                   || (nExtraRow != 0)
107                      && (row >= chunkSize - nExtraRow
108                         && split>=splitSize-1))
109                             continue;
110
111                      if(( *(Gka +  *(Maxka + row))).z >
112                           *( Da + chunkSize * row + column )){
113
114                        ( *(Gka + *(Maxka + row))).z
115                          = *(Da + chunkSize * row + column);
116
117                        ( *(Gka + *(Maxka + row))).x
118                          = split * chunkSize + row;
119
120                        ( *(Gka + *(Maxka + row))).y
121                          = chunk * chunkSize + column;
122
123
124                       /* Search the index of the max element
125                          within range */
126                       for (int index = beginSearch;
127                            index < endSearch;
128                            index ++){
129
130                            if((*(Gka+index)).z >
131                               (*(Gka + *(Maxka + row))).z)
132                                   *(Maxka + row) = index;
133                           } // inner for loop
134
135                       } // if
136
137          }   // (else condition) i.e., if split != chunk
138
139       }   // for loop ends
140
141     }   // if row < chunkSize
142
143  }
```

Data Structures

The input data set is represented in the form of a matrix, where each row represents a point and the respective columns represent the dimensions of the point. The complete input matrix contains n_{row} number of rows and n_{col} number of columns. Since, CUDA programming API does not support transferring of multidimensional arrays from host to device memory, we store the input matrix in a single dimensional array In of length $(n_{row} \times n_{col})$, the distance matrix chunks in a single dimensional array D of length $(n_{chunksize} \times n_{chunksize})$, given a fixed chunk size, $n_{chunksize}$ and the resultant kNN graph (Gk) in an array of 3-tuples $\{source, target, weight\}$ of length $(k \times n_{row})$. Additionally, we store the location of the farthest k nearest neighbours for each row index and chunk in an array $Maxk$ to facilitate the kNN search.

```
1
2  /* An array of 3-tuples {source,target,weight} to store the kNN graph (Gka),
3   this can  be replaced by a structure such as
4
5          typedef struct
6          {
7                  int source;
8                  int target;
9                  float weight;
10         } Edge;
11         Edge *Gka;
12
13         */
14
15         float3 *Gka;
16
17         float3 *Gka_sub; // A pointer to Gka (for a segment)
18
19         float *Ina; // Holds the input matrix
20         float *Da;  // Holds chunk of the original distance matrix
21
22         int k; // user defined value of k
23         int chunkSize;  // user defined chunk size
24         int *Maxka; // An array to hold the farthest value of k
25
26         int nRow; //Total Number rows in the input matrix
27         int nCol; //Total Number columns in the input matrix
```

Program Execution

We show a sample demonstration, input and output of our program in this section.

Input file format

Filename:Microarrays_CIBM_format.txt

```
<MicroarrayData>
10      6
F_1     1       4.5     2       7.3     8.21    5.71
F_2     1       3       8       6.66    7.19    1.06
F_3     3       3       9       5       6.67    7.73
F_4     4       4       1       4       3.64    2.92
F_5     8       7       1.5     4       4.91    6.15
F_6     8       6       3       2.4     7.36    4.76
F_7     3       8       3       3.2     5.31    5.49
F_8     2       1       4.02    5.8     3.08    7.15
F_9     4       1       3.7     2.2     1.31    1.48
F_10    -3      0       3       1       3.82    2.7
<SamplesNames>
        C1      C2      C3      C4      C5      C6
<SamplesClasses>
        1       1       1       0       0       0
<EndOfFile>
```

Running the Program

```
@mendel:./gpufsknn Microarrays_CIBM_format.txt

File Name (input)= TestData.bio.csv

Calculation starts at time : 17:15:49
Features (nRow) = 10  Samples(nCol) = 6
Data file loaded successfully...!
Type = Microarray
number of host CPUs:    16
number of CUDA devices: 1
   0: Tesla C2050
---------------------------
Total no of chunks in each split (splitSize):   1
Total no of splits in each segment (segmentSize):       1
CPU thread 0 (of 1) uses CUDA device 0
kNN computation finished at time : 17:15:55
Writing....output File Name = Microarrays_CIBM_format.txt.knn
```

Output file: Microarrays_CIBM_format.txt.knn

```
10 30
0 5 -0.145362
0 8 -0.782624
0 4 -0.117918
1 5 -0.480781
1 4 -0.866238
1 3 -0.415669
2 5 -0.486956
2 4 -0.763127
2 3 -0.833721
3 9 -0.520289
3 1 -0.415669
3 2 -0.833721
4 9 -0.673579
4 1 -0.866238
4 2 -0.763127
5 7 -0.620824
5 1 -0.480781
5 2 -0.486956
6 7 -0.317340
6 1 -0.292388
6 8 -0.834691
7 4 -0.367099
7 5 -0.620824
7 6 -0.317340
8 0 -0.782624
8 6 -0.834691
8 9 -0.426985
9 8 -0.426985
9 4 -0.673579
9 3 -0.520289
```

Conclusion

We developed a software tool based on our scalable approach for computing large-scale kNN graph using GPU and CUDA (FS-kNN-GPU). The source code is available under GNU Public License (GPL) at `https://sourceforge.net/p/gpufsknn/`

This is a hybrid approach where the host is responsible for loading, chunking and distributing the data and computation to the device, where the device uses two kernels to compute the distance and kNNs of each feature in the data set. The basic approach is simple and adaptable with other parallel frameworks. Outcome of our proposed tool can be used to generate approximate minimum spanning trees (AMST), minimum spanning forests (MSFs) [4] or clusters from large-scale biological data sets, such as microarrays (See [3]). We believe this work can provide compelling benefits for several domains of bioinformatics data mining.

References

[1] Chang D, Jones NA, Li D, Ouyang M, Ragade RK: Compute pairwise Euclidean distances of data points with GPUs. In *Proc. of the IASTED International Symposium on Computational Biology and Bioinformatics*, IASTED 2008:278–283.

[2] Chang D, Desoky AH, Ouyang M, C RE: Compute pairwise Euclidean distances of data points with GPUs. In *Proc. of the 10th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ACIS 2009:501–506.

[3] Arefin AS, Inostroza-Ponta M, Mathieson L, Berretta R, Moscato P: Clustering Nodes in Large-Scale Biological Networks Using External Memory Algorithms. In *ICA3PP (2)*, *Volume 7017 of* Lecture Notes in Computer Science. Edited by Xiang Y, Cuzzocrea A, Hobbs M, Zhou W, Springer 2011:375–386.

[4] Arefin AS, Riveros C, Berretta R, Moscato P (2012) knn-borvka-gpu: A fast and scalable mst construction from knn graphs on gpu. In: Murgante B, Gervasi O, Misra S, Nedjah N, Rocha AMAC, et al., editors, ICCSA (1). Springer, volume 7333 of *Lecture Notes in Computer Science*, pp. 71-86.

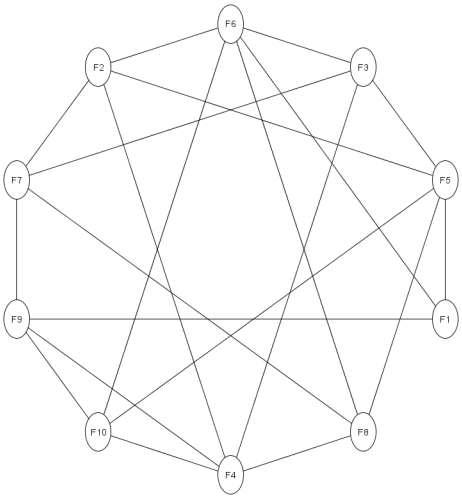

Figure 1: Results can be visualized using yEd (http://www.yworks.com) GML visualization tool. Visualization of the 3NN graph generated by the input file.

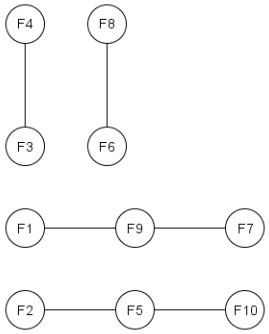

Figure 2: Visualization of the clustering created using the kNN graph and the MSTkNN+ algorithm in [3].