3 CUDA Implementations of the kNN Algorithm

July 23, 2019

Contents

Introduction

kNN Algorithm Description

Let $R = \{r_1, r_2, ..., r_m\}$ be a set of m reference points with values in R^d, and let $Q = \{q_1, q_2, ... ,q_n\}$ be a set of n query points in the same space. The kNN search problem consists in searching the k nearest neighbors of each query point $q_i \in$ Q in the reference set R given a specific distance. Commonly, the Euclidean or the Manhattan distance is used but any other distance can be used (for example, the Chebyshev norm or the Mahalanobis distance).

The "brut-force" KNN algorithm goes as following:

1. Compute all the distances between q_i and r_j, $j \in [1, m]$.
2. Sort the computed distances.
3. Select the k reference points corresponding to the k smallest distances.
4. Repeat steps 1. to 3. for all query points.

If we apply this algorithm for the n query points and consider the typical case of large sets (both references and queries), the complexity of this algorithm is $O(nmd)$ multiplications for the n $\times$ m distances computed and $O(nmlogm)$ for the n sorting processes. However, this algorithm is parallelizable and suitable for a GPU implementation.

Sorting Algorithm

A popular sorting algorithm Quicksort is one of the fastest sorting algorithms in practice. However, it is recursive and CUDA does not allow recursive functions. The comb sort complexity is $O(nlogn)$ both in the worst and average cases. It is also among the fastest algorithms and simple to implement. Nevertheless, keeping in mind that we are only interested in the k smallest elements, k being usually small, we use an insertion sort variant which only outputs the k smallest elements.

CUDA Implementation

The implementation is composed of two kernels (CUDA functions):

1. The first kernel computes the distance matrix of size $m \times n$ containing the distances between the n query points and the m reference points. The computation of this matrix was fully parallelized since the distances between pairs of points are independent: each thread computed the distance between a given query point q_i and a given reference point r_j. Two kinds of memory are used: global memory and texture memory. The global memory has a huge bandwidth but the performances decrease if the memory accesses are non-coalesced. In this case, the texture memory is a good option because there are less penalties for non-coalesced readings. In one of our implementations, we use global memory for storing the query set (coalesced readings), and texture memory for the reference set (non-coalesced readings).

2. The second kernel sortes the distance matrix. The n sorting processes (one for each query point) were parallelized since they are independent: each thread sorted all the distances computed for a given query point. The sorting consists in comparing and exchanging many distances in a non-predictable order. Therefore, the memory accesses are not coalesced, indicating that the texture memory could be appropriate. But it is a read-only memory. Only the global memory allows readings and writings. This penalizes the sorting performance.

In our implementation, the sorting algorithm used was a modified version of the insertion sort. From experiments, for small values of k, this sorting algorithm appears to be faster than the efficient comb sort algorithm.

In addition, to store the indices of the k-nearest neighbors, an index matrix of size $m \times n$ containing and on each column the indices of the reference points per query were defined. Each column was initialized with the vector $(1, 2, . . . \mathrm{m})\top$, where $\mathrm{M}\top$ denotes the transpose of M . The element insertion performed in the distance matrix as part of the sorting processes was simultaneously applied to the index matrix so that, in the end, its uppermost $k \times n$-submatrix corresponded to the queries k-nearest neighbor indices ordered by increasing distance. Working with an initial $m \times n$-index matrix is a waste of memory. We work only with a $k \times n$-index matrix from the beginning, thus avoiding the $(\mathrm{m} - \mathrm{k}) \times \mathrm{n}$ memory overhead. The sorting process deals with each array column monotonically from the first to the last element. Consequently, at each iteration, the index of the considered reference point is known. While sorting, if the l-th element needs to be inserted into the distance matrix, the index value l is also inserted into the $k \times n$-index matrix at the exact same position, iteratively filling in the whole matrix.

CUBLAS Implementation

BLAS is a linear algebra library specialized in vector/matrix operations. CUBLAS is the CUDA implementation of BLAS. The distance matrix computation represents the main part of the computation time in the kNN algorithm. Here, we show that we can improve the global performances of the kNN search by reformulating the way to compute the distance matrix and by using the

CUBLAS library. Let us consider two points x and y in a d-dimensional space $x = (x1, x2, xd)^\top$, $y = (y1, y2, yd)^\top$, where M⊤ denotes the transpose of M. The classical Euclidian distance computation can be rewritten using matrix additions and multiplications:
$\rho^2(x, y) = (x - y)^\top(x - y) = \| x \|^2 + \| y \|^2 - 2x^\top y$, where $\|.\|$ is the Euclidean norm. The square root is then computed at the end. This approach can be extended to handle sets of points. Let R and Q be two matrices of size $d \times m$ and $d \times n$, respectively, containing the m reference points and the n query points, respectively. The $m \times n$-matrix $\rho^2(R, Q)$ containing all the pairwise squared distances between query points and reference points is given by
$\rho^2(R, Q) = N_R + N_Q - 2R^\top Q$. The elements of the ith row of N_R are all equal to $\| r_i \|^2$. The elements of the j th column of N_Q are all equal to $\| q_j \|^2$. The way $\rho^2(R, Q)$is expressed (i.e., through matrix additions and multiplications) is perfectly adapted to a CUBLAS implementation (only N_R and N_Q have to be computed separately). However, this method is highly demanding: N_R, N_Q , and $R^\top Q$ are stored in three matrices of size m × n. Nevertheless, the matrices N_R and N_Q have a specific form. To optimize the memory usage, N_R and N_Q were stored as vectors of dimension $1 \times m$ and $1 \times n$, respectively. The ith element of N_R is equal to $\| r_i \|^2$, and similarly for N_Q. Then, the addition and subtraction were handled by CUDA kernels. The proposed kNN search implementation is composed of the following kernels:
1. Compute the vector N_R using CUDA (coalesced read/write);

2. Compute the vector N_Q using CUDA (coalesced read/write);

3. Compute the $m \times n$-matrix $A = -2R^\top Q$ using CUBLAS;

4. Add the ith element of N_R to every element of the ith row of the matrix A using CUDA (grid of $m \times n$ threads, non coalesced read/write: use of the shared memory); The resulting matrix is denoted by B;

5. Sort in parallel each column of B (with n threads) using the modified insertion sort. The resulting matrix is denoted by C;

6. Add the jth value of N_Q to the first k elements of the jth column of the matrix C using CUDA (coalesced read/write). The resulting matrix is denoted by D;

7. Compute the square root of the first k elements of D to obtain the k smallest distances (coalesced read/write). The resulting matrix is denoted by E ;

8. Extract the uppermost $k \times n$-submatrix of E; The resulting matrix is the distance matrix for the k-nearest neighbors of each query.

Note that the matrix names were given for algorithmic clarity only. In reality, once A is computed, all the remaining computations are done "in place", meaning that the matrices from A to E are in fact a single matrix occupying a unique area of memory. If the matrix A does not fit into the GPU memory, the query points are splitted, processed separately, and the distances to the k-nearest neighbors are then merged together on the CPU/classical memory side. The indices of the k-nearest neighbors were stored in an index matrix I of size $k \times n$.

About CUDA

At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are exposed as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They allow to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition allows threads to cooperate when solving each sub-problem and enables automatic scalability. Each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count.

Kernels

CUDA C allows to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA threads. A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using <<< ... >>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

Thread Hierarchy

threadIdx is a 3-component vector which forms a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*.

The index of a thread and its thread ID relate to each other in a straightforward way: for a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy),the thread ID of a thread of index (x, y) is $(x + yDx)$; for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is $(x + yDx + zDxDy)$.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

A kernel can be executed by multiple equally-shaped thread blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid. The number of threads per block and the number of blocks per grid specified in the <<< ... >>> syntax can be of type int or dim3.

Each block within the grid can be identified by an index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. One can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Heterogeneous Programming

The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. The CUDA programming model assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Unified Memory provides managed memory to bridge the host and device memory spaces. Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space.

Device Memory Accesses

An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp.

Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. To maximize global memory throughput, it is therefore important to maximize coalescing by: Using data types that meet the size and alignment requirement Padding data in some cases, for example, when accessing a two-dimensional array

Size and Alignment Requirement

Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size). If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use types that meet this requirement for data that resides in global memory.

The alignment requirement is automatically fulfilled for the built-in types of char, short, int, long, longlong, float, double like float2 or float4. For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers `__align__`(8) or `__align__`(16).

Shared Memory

Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests.

Texture and Surface Memory

The texture and surface memory spaces reside in device memory and are cached in texture cache, so a texture fetch or surface read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance.

Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

1. If the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads;

2. Addressing calculations are performed outside the kernel by dedicated units;

3. Packed data may be broadcast to separate variables in a single operation;

4. 8-bit and 16-bit integer input data may be optionally converted to 32 bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0]

Compiler Overview

The compiler, called nvcc, is part of NVIDIA's production CUDA toolchain. The heterogeneous CUDA program containing host and device code is input to a CUDA C language front end (CUDAFE). The front end partitions the program into a host part and a device part. The host part is compiled as C++ code by the host compiler and the device part is fed to a high-level backend based on Open64 [10] and targets the PTX instruction set. The PTX code is compiled by a device specific optimizing code generator called PTXAS. The compiled host code is combined with the device code to create an executable application.

kNN CUDA Global Implementation

Defining the Algorithm

The most time-consuming part of kNN includes the distance calculation component and the sorting component. Therefore, GPU-based implementations focus on accelerating these two components. Here, we show three implementations of the kNN algorithm: `knn_cuda_global`, `knn_cuda_texture`, `knn_cublas`.

`knn_cuda_global` computes the k-NN using the GPU global memory for storing reference and query points, distances and indexes.

`knn_cuda_texture` computes the k-NN using the GPU texture memory for storing the reference points and the GPU global memory for storing other arrays. Using a texture usually speeds-up the computations compared to the first approach. However, due to a size constraint of the texture structures in CUDA, this implementation might not an option for some high dimensional problems.

`knn_cublas`computes the k-NN using a different technique involving the use of CUBLAS (CUDA implementation of BLAS). The computation of the distances are split into several sub-problems better suited to a GPU acceleration. On specific problems, this implementation may lead to a faster processing time compared to the first two approaches.

We start by implementing the first version of the kNN algorithm, `knn_cuda_global`.

We first import some libraries:

[0]:
```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <cublas.h>
#include <map>
#include <utility>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
```

```
#include <algorithm>

#define BLOCK_DIM 16
```

We now define a function that calculates the squared Euclidian distance matrix between the query and the reference points. The query and reference points are stored in the GPU global memory in this case. The input matrices containing query and reference points are split into sub-matrices, and they (sub-matrix of reference points and sub-matrix of query points) are stored in the shared memory of a particular thread block. The calculation of distances is performed in sub-matrices in parallel in a number of thread blocks.

In the function, we first declare the shared memory arrays to store the sub-matrix of reference (A) and sub-matrix of query points (B). We then initialize and specify parameters of a particular sub-matrix A (begin, step, end): the start index for each sub-matrix, the step to compute the address of an array's element, and the end index. Analogously, we initialize and specify parameters of a particular sub-matrix B (begin, step). Note that given the row and column of an array element of type T, the address of the element is computed as: T * pElement = (T *)((*char*)BaseAddress + Row * pitch) + Column.

Next, we initialize SSD for the current thread and obtain thread index (x,y). Notice that both input matrices A and B are organized in the transpose format in the GPU device memory, whose columns correspond to the related data points. Because the blocks may exceed the border of the input matrices, we check several checking conditions to ensure zeros are used when the blocks run out of the boundary to acquire correct results. Next, in the loop, we load the matrices from device memory to shared memory; each thread loads one element of each matrix. All threads are synchronized to the same stage when data have been loaded into the shared memory from the device memory. Then feature differences are computed between any two threads and accumulated to the related result. All threads are synchronized again before moving to the next iteration of the loop. When all features have been scanned through by every thread, the final distance is calculated and stored to the corresponding position in the distance matrix based on the current block index and thread index. The distance matrix is then returned at the end of the kernel function. At this point, the distance matrix (*nxm* matrix where *n* is the number of reference points and *m* is the number of query points) is still stored on the GPU device memory and it must be copied back to the system memory for any future processing by CPU. By using shared memory, the threads can avoid frequently accessing device memory to fetch data so that better performance is achieved.

[0]:
```
/*
 The function has the following input parameters:
 * param ref:          refence points stored in the global memory
 * param ref_width:    number of reference points
 * param ref_pitch:    pitch of the reference points array in number of columns
 * param query:        query points stored in the global memory
 * param query_width:  number of query points
 * param query_pitch:  pitch of the query points array in number of columns
 * param height:       dimension of points = height of the array `ref` and of␣
 ↪the array `query`
 * param dist:         array containing the query_width x ref_width computed␣
 ↪distances
 */
__global__ void compute_distances(float * ref,
                                  int     ref_width,
                                  int     ref_pitch,
                                  float * query,
                                  int     query_width,
                                  int     query_pitch,
                                  int     height,
                                  float * dist) {

    // Declaration of the shared memory arrays As and Bs used to store the␣
 ↪sub-matrix of A and B
    __shared__ float shared_A[BLOCK_DIM][BLOCK_DIM];
    __shared__ float shared_B[BLOCK_DIM][BLOCK_DIM];

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initializarion of the SSD for the current thread
    float ssd = 0.f;


    // Sub-matrix of A (begin, step, end) and Sub-matrix of B (begin, step)
    __shared__ int begin_A;
    __shared__ int begin_B;
    __shared__ int step_A;
    __shared__ int step_B;
    __shared__ int end_A;


    // Loop parameters
    begin_A = BLOCK_DIM * blockIdx.y;
    begin_B = BLOCK_DIM * blockIdx.x;

    step_A  = BLOCK_DIM * ref_pitch; //ref_pitch is the (potentially padded to␣
 ↪satisfy size and alignment requirements) width of the array that stores␣
 ↪reference points
    step_B  = BLOCK_DIM * query_pitch; //query_pitch is the (potentially padded␣
 ↪to satisfy size and alignment requirements) width of the array that stores␣
 ↪query points

    end_A   = begin_A + (height-1) * ref_pitch;

    // Conditions
    int cond0 = (begin_A + tx < ref_width); // used to write in shared memory //
 ↪stay within A matrix on the X axis
```

```
   int cond1 = (begin_B + tx < query_width); // used to write in shared memory␣
↪& to computations and to write in output array  //stay within B matrix on the␣
↪X axis
   int cond2 = (begin_A + ty < ref_width); // used for computations and to␣
↪write in output matrix //stay within A matrix in Y direction?

   // Loop over all the sub-matrices of A and B required to compute the block␣
↪sub-matrix
   for (int a = begin_A, b = begin_B; a <= end_A; a += step_A, b += step_B) {

       // Load the matrices from device memory to shared memory; each thread␣
↪loads one element of each matrix

       //conditoin to check if y coordinate of an element is less than matrix␣
↪height (i.e. if this element belongs to this matrix)
       if (a/ref_pitch + ty < height) {
           shared_A[ty][tx] = (cond0)? ref[a + ref_pitch * ty + tx] : 0; //␣
↪shared_A[ty][tx] will take the value of ref[a + ref_pitch * ty + tx] if cond0␣
↪is True; 0 otherwise
           shared_B[ty][tx] = (cond1)? query[b + query_pitch * ty + tx] : 0;
       }
       else {
           shared_A[ty][tx] = 0;
           shared_B[ty][tx] = 0;
       }

       // Synchronize to make sure the matrices are loaded
       __syncthreads();

       // Compute the difference between the two matrixes; each thread computes␣
↪one element of the block sub-matrix
       if (cond2 && cond1) {
           for (int k = 0; k < BLOCK_DIM; ++k){
               float tmp = shared_A[k][ty] - shared_B[k][tx];
               ssd += tmp*tmp;
           }
       }

       // Synchronize to make sure that the preceeding computation is done␣
↪before loading two new sub-matrices of A and B in the next iteration
       __syncthreads();
   }

   // Write the block sub-matrix to device memory; each thread writes one␣
↪element
   if (cond2 && cond1) {
        dist[ (begin_A + ty) * query_pitch + begin_B + tx ] = ssd;
    }
}
```

We then define a function that for each query point (i.e. each column) finds the k-th smallest distances of the distance matrix and their respective indexes and gathers them at the top of the 2 arrays. Since we only need to locate the k smallest distances, sorting the entire array would not be very efficient if k is relatively small. Here, we perform a simple insertion sort by eventually inserting a given distance in the first k values.

[0]:
```
/*
 * @param dist          distance matrix
 * @param dist_pitch    pitch of the distance matrix given in number of columns
 * @param index         index matrix
 * @param index_pitch   pitch of the index matrix given in number of columns
 * @param width         width of the distance matrix and of the index matrix
 * @param height        height of the distance matrix
 * @param k             number of values to find
 */
__global__ void modified_insertion_sort(float * dist,
                                        int     dist_pitch,
                                        int *   index,
                                        int     index_pitch,
                                        int     width,
                                        int     height,
                                        int     k){

    // Column position
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

    // Do nothing if we are out of bounds
    if (xIndex < width) {

        // Pointer shift
        float * p_dist  = dist  + xIndex;
        int *   p_index = index + xIndex;

        // Initialise the first index
        p_index[0] = 0;

        // Go through all points
        for (int i=1; i<height; ++i) {

            // Store current distance and associated index
            float curr_dist = p_dist[i*dist_pitch];
            int   curr_index  = i;
            // Skip the current value if its index is >= k and if it's higher␣
 ↪the k-th already sorted smallest value
            if (i >= k && curr_dist >= p_dist[(k-1)*dist_pitch]) {
                continue;
            }

            // Shift values (and indexes) higher than the current distance to␣
 ↪the right
            int j = min(i, k-1);
            while (j > 0 && p_dist[(j-1)*dist_pitch] > curr_dist) {
                p_dist[j*dist_pitch]   = p_dist[(j-1)*dist_pitch];
                p_index[j*index_pitch] = p_index[(j-1)*index_pitch];
                --j;
            }

            // Write the current distance and index at their position
            p_dist[j*dist_pitch]   = curr_dist;
            p_index[j*index_pitch] = curr_index;
        }
    }
}
```

We now define a function that computes the square root of the first k lines of the distance matrix.

[0]:
```
/*
 * @param dist   distance matrix
 * @param width  width of the distance matrix
 * @param pitch  pitch of the distance matrix given in number of columns
 * @param k      number of values to consider
 */
__global__ void compute_sqrt(float * dist, int width, int pitch, int k){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
    if (xIndex<width && yIndex<k)
        dist[yIndex*pitch + xIndex] = sqrt(dist[yIndex*pitch + xIndex]);
}
```

Now, we are ready to put all the kernels defined above into one main function called knn_cuda_global. In the function, we first check that there is a CUDA device available and that we can select it. We then allocate global memory initializing pointers to the reference points matrix, query points matrix, distance matrix, index matrix as well as for the reference points matrix pitch in bytes, query points matrix pitch in bytes, distance matrix pitch in bytes and index matrix pitch in bytes. The pitch values in bytes are used to allocate memory for this matrices so that these matrices satisfy the size and alignment requirement. If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use types that meet this requirement for data that resides in global memory.

Next, we deduce pitch values from bytes into the number of columns and check that the pitch values are valid by ensuring that the pitch value of query points matrix is the same as that of the distance matrix and the same as that of the index matrix. We then copy the data from the host to the device. Next, we launch the kernel to compute the squared Euclidean distances. We then launch the kernel that sorts the distances with their respective indexes as well as the kernel that computes the square root of the k smallest distances. Finally, we copy k smallest distances / indexes from the device to the host and free the memory on the device and on the host.

[0]:
```
bool knn_cuda_global(const float * ref,
                     int           ref_nb,
                     const float * query,
                     int           query_nb,
                     int           dim,
                     int           k,
                     float *       knn_dist,
                     int *         knn_index) {

    // Constants
    const unsigned int size_of_float = sizeof(float);
    const unsigned int size_of_int   = sizeof(int);

    // Return variables
    cudaError_t err0, err1, err2, err3;

    // Check that we have at least one CUDA device
    int nb_devices;
    err0 = cudaGetDeviceCount(&nb_devices);
    if (err0 != cudaSuccess || nb_devices == 0) {
        printf("ERROR: No CUDA device found\n");
        return false;
    }

    // Select the first CUDA device as default
    err0 = cudaSetDevice(0);
    if (err0 != cudaSuccess) {
        printf("ERROR: Cannot set the chosen CUDA device\n");
        return false;
    }

    // Allocate global memory
    float * ref_dev   = NULL;
    float * query_dev = NULL;
    float * dist_dev  = NULL;
    int   * index_dev = NULL;
    size_t  ref_pitch_in_bytes;
    size_t  query_pitch_in_bytes;
    size_t  dist_pitch_in_bytes;
    size_t  index_pitch_in_bytes;
    err0 = cudaMallocPitch((void**)&ref_dev,   &ref_pitch_in_bytes,   ref_nb   *␣
↪size_of_float, dim);
    err1 = cudaMallocPitch((void**)&query_dev, &query_pitch_in_bytes, query_nb *␣
↪size_of_float, dim);
    err2 = cudaMallocPitch((void**)&dist_dev,  &dist_pitch_in_bytes,  query_nb *␣
↪size_of_float, ref_nb);
    err3 = cudaMallocPitch((void**)&index_dev, &index_pitch_in_bytes, query_nb *␣
↪size_of_int,   k);
    if (err0 != cudaSuccess || err1 != cudaSuccess || err2 != cudaSuccess ||␣
↪err3 != cudaSuccess) {
        printf("ERROR: Memory allocation error\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Deduce pitch values
    size_t ref_pitch   = ref_pitch_in_bytes   / size_of_float;
    size_t query_pitch = query_pitch_in_bytes / size_of_float;
    size_t dist_pitch  = dist_pitch_in_bytes  / size_of_float;
    size_t index_pitch = index_pitch_in_bytes / size_of_int;

    // Check pitch values
    if (query_pitch != dist_pitch || query_pitch != index_pitch) {
        printf("ERROR: Invalid pitch value\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Copy reference and query data from the host to the device
    err0 = cudaMemcpy2D(ref_dev,   ref_pitch_in_bytes,   ref,   ref_nb *␣
↪size_of_float,   ref_nb * size_of_float,   dim, cudaMemcpyHostToDevice);
    err1 = cudaMemcpy2D(query_dev, query_pitch_in_bytes, query, query_nb *␣
↪size_of_float, query_nb * size_of_float, dim, cudaMemcpyHostToDevice);
    if (err0 != cudaSuccess || err1 != cudaSuccess) {
        printf("ERROR: Unable to copy data from host to device\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Compute the squared Euclidean distances
    dim3 block0(BLOCK_DIM, BLOCK_DIM, 1);
    dim3 grid0(query_nb / BLOCK_DIM, ref_nb / BLOCK_DIM, 1);
    if (query_nb % BLOCK_DIM != 0) grid0.x += 1;
    if (ref_nb   % BLOCK_DIM != 0) grid0.y += 1;
    compute_distances<<<grid0, block0>>>(ref_dev, ref_nb, ref_pitch, query_dev,␣
↪query_nb, query_pitch, dim, dist_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Sort the distances with their respective indexes
    dim3 block1(256, 1, 1);
    dim3 grid1(query_nb / 256, 1, 1);
    if (query_nb % 256 != 0) grid1.x += 1;
    modified_insertion_sort<<<grid1, block1>>>(dist_dev, dist_pitch, index_dev,␣
↪index_pitch, query_nb, ref_nb, k);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Compute the square root of the k smallest distances
    dim3 block2(16, 16, 1);
    dim3 grid2(query_nb / 16, k / 16, 1);
    if (query_nb % 16 != 0) grid2.x += 1;
    if (k % 16 != 0)        grid2.y += 1;
    compute_sqrt<<<grid2, block2>>>(dist_dev, query_nb, query_pitch, k);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }
    // Copy k smallest distances / indexes from the device to the host
    err0 = cudaMemcpy2D(knn_dist,  query_nb * size_of_float, dist_dev, ␣
↪dist_pitch_in_bytes,  query_nb * size_of_float, k, cudaMemcpyDeviceToHost);
    err1 = cudaMemcpy2D(knn_index, query_nb * size_of_int,   index_dev,␣
↪index_pitch_in_bytes, query_nb * size_of_int,   k, cudaMemcpyDeviceToHost);
    if (err0 != cudaSuccess || err1 != cudaSuccess) {
        printf("ERROR: Unable to copy data from device to host\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Memory clean-up
    cudaFree(ref_dev);
    cudaFree(query_dev);
    cudaFree(dist_dev);
    cudaFree(index_dev);

    return true;
}
```

kNN CUDA Texture Implementation

Here, we present another implementation of the kNN algorithm in which the GPU texture memory is used for storing reference points.

We first define compute_distances function which calculates the distances between the query and reference points. The GPU texture memory is used for storing the reference points, and the GPU global memory is used for storing other arrays. Using a texture usually speeds-up the computations compared to the first approach. Like constant memory, texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality. However, due to a size constraint of the texture structures in CUDA, this implementation might not an option for some high dimensional problems. The texture memory space resides in device memory and is cached in texture cache, so a texture fetch read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance.

Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

1. If the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads;

2. Addressing calculations are performed outside the kernel by dedicated units;
3. Packed data may be broadcast to separate variables in a single operation;
4. 8-bit and 16-bit integer input data may be optionally converted to 32 bit floating- point values in the range [0.0, 1.0] or [-1.0, 1.0]

[0]:
```
/**
 * Computes the squared Euclidean distance matrix between the query points and␣
 ↪the reference points.
 *
 * @param ref          refence points stored in the texture memory
 * @param ref_width    number of reference points
 * @param query        query points stored in the global memory
 * @param query_width  number of query points
 * @param query_pitch  pitch of the query points array in number of columns
 * @param height       dimension of points = height of texture `ref` and of the␣
 ↪array `query`
 * @param dist         array containing the query_width x ref_width computed␣
 ↪distances
 */
__global__ void compute_distance_texture(cudaTextureObject_t ref,
                                         int                 ref_width,
                                         float *             query,
                                         int                 query_width,
                                         int                 query_pitch,
                                         int                 height,
                                         float*              dist) {

    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

    if ( xIndex<query_width && yIndex<ref_width) {
        float ssd = 0.f;
        for (int i=0; i<height; i++) {
            float tmp  = tex2D<float>(ref, (float)yIndex, (float)i) - query[i *␣
 ↪query_pitch + xIndex];
            ssd += tmp * tmp;
        }
        dist[yIndex * query_pitch + xIndex] = ssd;
    }
}
```

Now, we are ready to define our main function called knn_cuda_texture.

In the function, we first check that there is a CUDA device available and that we can select it. We then allocate global memory initializing pointers to the query points matrix, distance matrix, index matrix as well as for the query points matrix pitch in bytes, distance matrix pitch in bytes and index matrix pitch in bytes. The pitch values in bytes are used to allocate memory for this matrices so that it satisfies the size and alignment requirement. If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. Next, we deduce pitch values from bytes into the number of columns and check that the pitch values are valid by ensuring that the pitch value of query points matrix is the same as that of the distance matrix and the same as that of the index matrix. We then copy data from the host to the device. After that, we allocate CUDA array for reference points and check that the allocation was succesful. We then copy the reference points from host to device. Next, we specify the Resource descriptor and Texute descriptor and create the texture. Next, we launch the kernel to compute the squared Euclidean distances. We then launch the kernel that sorts the distances with their respective indexes as well as the kernel that computes the square root of the k smallest distances. Finally, we copy k smallest distances / indexes from the device to the host and free the memory on the device and on the host.

[0]:
```
bool knn_cuda_texture(const float * ref,
                      int           ref_nb,
                      const float * query,
                      int           query_nb,
                      int           dim,
                      int           k,
                      float *       knn_dist,
                      int *         knn_index) {

    // Constants
    unsigned int size_of_float = sizeof(float);
    unsigned int size_of_int   = sizeof(int);

    // Return variables
    cudaError_t err0, err1, err2;

    // Check that we have at least one CUDA device
    int nb_devices;
    err0 = cudaGetDeviceCount(&nb_devices);
    if (err0 != cudaSuccess || nb_devices == 0) {
        printf("ERROR: No CUDA device found\n");
        return false;
    }

    // Select the first CUDA device as default
    err0 = cudaSetDevice(0);
    if (err0 != cudaSuccess) {
        printf("ERROR: Cannot set the chosen CUDA device\n");
        return false;
    }

    // Allocate global memory
    float * query_dev = NULL;
    float * dist_dev  = NULL;
    int *   index_dev = NULL;
    size_t  query_pitch_in_bytes;
    size_t  dist_pitch_in_bytes;
    size_t  index_pitch_in_bytes;
    err0 = cudaMallocPitch((void**)&query_dev, &query_pitch_in_bytes, query_nb *␣
↪size_of_float, dim);
    err1 = cudaMallocPitch((void**)&dist_dev,  &dist_pitch_in_bytes,  query_nb *␣
↪size_of_float, ref_nb);
    err2 = cudaMallocPitch((void**)&index_dev, &index_pitch_in_bytes, query_nb *␣
↪size_of_int,   k);
    if (err0 != cudaSuccess || err1 != cudaSuccess || err2 != cudaSuccess) {
        printf("ERROR: Memory allocation error (cudaMallocPitch)\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Deduce pitch values
    size_t query_pitch = query_pitch_in_bytes / size_of_float;
    size_t dist_pitch  = dist_pitch_in_bytes  / size_of_float;
    size_t index_pitch = index_pitch_in_bytes / size_of_int;

    // Check pitch values
    if (query_pitch != dist_pitch || query_pitch != index_pitch) {
        printf("ERROR: Invalid pitch value\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Copy query data from the host to the device
    err0 = cudaMemcpy2D(query_dev, query_pitch_in_bytes, query, query_nb *␣
↪size_of_float, query_nb * size_of_float, dim, cudaMemcpyHostToDevice);
    if (err0 != cudaSuccess) {
        printf("ERROR: Unable to copy data from host to device\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Allocate CUDA array for reference points
    cudaArray* ref_array_dev = NULL;
    cudaChannelFormatDesc channel_desc = cudaCreateChannelDesc(32, 0, 0, 0,␣
↪cudaChannelFormatKindFloat);
    err0 = cudaMallocArray(&ref_array_dev, &channel_desc, ref_nb, dim);
    if (err0 != cudaSuccess) {
        printf("ERROR: Memory allocation error (cudaMallocArray)\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        return false;
    }

    // Copy reference points from host to device
    err0 = cudaMemcpyToArray(ref_array_dev, 0, 0, ref, ref_nb * size_of_float *␣
↪dim, cudaMemcpyHostToDevice);
    if (err0 != cudaSuccess) {
        printf("ERROR: Unable to copy data from host to device\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        return false;
    }

    // Resource descriptor
    struct cudaResourceDesc res_desc;
    memset(&res_desc, 0, sizeof(res_desc));
    res_desc.resType           = cudaResourceTypeArray;
    res_desc.res.array.array = ref_array_dev;

    // Texture descriptor
    struct cudaTextureDesc tex_desc;
    memset(&tex_desc, 0, sizeof(tex_desc));
    tex_desc.addressMode[0]    = cudaAddressModeClamp;
    tex_desc.addressMode[1]    = cudaAddressModeClamp;
    tex_desc.filterMode        = cudaFilterModePoint;
    tex_desc.readMode          = cudaReadModeElementType;
    tex_desc.normalizedCoords = 0;

    // Create the texture
    cudaTextureObject_t ref_tex_dev = 0;
    err0 = cudaCreateTextureObject(&ref_tex_dev, &res_desc, &tex_desc, NULL);
    if (err0 != cudaSuccess) {
        printf("ERROR: Unable to create the texture\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        return false;
    }

    // Compute the squared Euclidean distances
    dim3 block0(16, 16, 1);
    dim3 grid0(query_nb / 16, ref_nb / 16, 1);
    if (query_nb % 16 != 0) grid0.x += 1;
    if (ref_nb   % 16 != 0) grid0.y += 1;
    compute_distance_texture<<<grid0, block0>>>(ref_tex_dev, ref_nb, query_dev,␣
↪query_nb, query_pitch, dim, dist_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        cudaDestroyTextureObject(ref_tex_dev);
        return false;
    }

    // Sort the distances with their respective indexes
    dim3 block1(256, 1, 1);
    dim3 grid1(query_nb / 256, 1, 1);
    if (query_nb % 256 != 0) grid1.x += 1;
    modified_insertion_sort<<<grid1, block1>>>(dist_dev, dist_pitch, index_dev,␣
↪index_pitch, query_nb, ref_nb, k);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        cudaDestroyTextureObject(ref_tex_dev);
        return false;
    }

    // Compute the square root of the k smallest distances
    dim3 block2(16, 16, 1);
    dim3 grid2(query_nb / 16, k / 16, 1);
    if (query_nb % 16 != 0) grid2.x += 1;
    if (k % 16 != 0)        grid2.y += 1;
    compute_sqrt<<<grid2, block2>>>(dist_dev, query_nb, query_pitch, k);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        cudaDestroyTextureObject(ref_tex_dev);
        return false;
    }
    // Copy k smallest distances / indexes from the device to the host
    err0 = cudaMemcpy2D(knn_dist,  query_nb * size_of_float, dist_dev, ␣
↪dist_pitch_in_bytes,  query_nb * size_of_float, k, cudaMemcpyDeviceToHost);
    err1 = cudaMemcpy2D(knn_index, query_nb * size_of_int,   index_dev,␣
↪index_pitch_in_bytes, query_nb * size_of_int,   k, cudaMemcpyDeviceToHost);
    if (err0 != cudaSuccess || err1 != cudaSuccess) {
        printf("ERROR: Unable to copy data from device to host\n");
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFreeArray(ref_array_dev);
        cudaDestroyTextureObject(ref_tex_dev);
        return false;
    }

    // Memory clean-up
    cudaFree(query_dev);
    cudaFree(dist_dev);
    cudaFree(index_dev);
    cudaFreeArray(ref_array_dev);
    cudaDestroyTextureObject(ref_tex_dev);

    return true;
}
```

kNN CUBLAS Implementation

In this implementation, we use CUBLAS library to calculate the distance matrix. CUBLAS is the official version of BLAS routine for GPU made by NVIDIA and it has proved to be extremely efficient compared to the same implementations on CPU, e.g. using MKL.

We first define a kernel that computes the squared norm of each column of the input array.

[0]:
```
/**
 * Computes the squared norm of each column of the input array.
 *
 * @param array   input array
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param height  number of rows of `array` = dimension of the points
 * @param norm    output array containing the squared norm values
 */
__global__ void compute_squared_norm(float * array, int width, int pitch, int␣
↪height, float * norm){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex<width){
        float sum = 0.f;
        for (int i=0; i<height; i++){
            float val = array[i*pitch+xIndex];
            sum += val*val;
        }
        norm[xIndex] = sum;
    }
}
```

We then define a kernel that adds the reference points norm (column vector) to each colum of the input array.

[0]:
```
/**
 * Add the reference points norm (column vector) to each colum of the input␣
 ↪array.
 *
 * @param array   input array
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param height  number of rows of `array` = dimension of the points
 * @param norm    reference points norm stored as a column vector
 */
__global__ void add_reference_points_norm(float * array, int width, int pitch,␣
 ↪int height, float * norm){
    unsigned int tx = threadIdx.x;
    unsigned int ty = threadIdx.y;
    unsigned int xIndex = blockIdx.x * blockDim.x + tx;
    unsigned int yIndex = blockIdx.y * blockDim.y + ty;
    __shared__ float shared_vec[16];
    if (tx==0 && yIndex<height)
        shared_vec[ty] = norm[yIndex];
    __syncthreads();
    if (xIndex<width && yIndex<height)
        array[yIndex*pitch+xIndex] += shared_vec[ty];
}
```

We now define a kernel that adds the query points norm (row vector) to the k first lines of the input array and computes the square root of the resulting values.

[0]:
```
/**
 * Adds the query points norm (row vector) to the k first lines of the input
 * array and computes the square root of the resulting values.
 *
 * @param array   input array
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param k       number of neighbors to consider
 * @param norm     query points norm stored as a row vector
 */
```

```
__global__ void add_query_points_norm_and_sqrt(float * array, int width, int␣
↪pitch, int k, float * norm){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
    if (xIndex<width && yIndex<k)
        array[yIndex*pitch + xIndex] = sqrt(array[yIndex*pitch + xIndex] +␣
↪norm[xIndex]);
}
```

Now, we can put the kernels defined above into one main function called knn_cublas.

In the function, we first check that there is a CUDA device available and that we can select it. We then allocate global memory initializing pointers to the reference points matrix, query points matrix, distance matrix, index matrix as well as for the reference points matrix pitch in bytes, query points matrix pitch in bytes, distance matrix pitch in bytes and index matrix pitch in bytes. The pitch values in bytes are used to allocate memory for this matrices so that it satisfies the size and alignment requirement. If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. Next, we deduce pitch values from bytes into the number of columns and check that the pitch values are valid by ensuring that the pitch value of query points matrix is the same as that of the distance matrix and the same as that of the index matrix. We then copy data from the host to the device and check that the copying was successful. Next, we launch the kernel to compute the squared norm for the reference points as well as the kernel to compute the squared norm for the query points. We then compute query * transpose(reference). We launch a kernel that adds reference points norm, then launch the kernel to sort each column as well as the kernel to add query norm and compute the square root of the k first elements. Finally, we copy k smallest distances / indexes from the device to the host and free the memory on the device and on the host.

[0]:
```
bool knn_cublas(const float * ref,
                int           ref_nb,
                const float * query,
                int           query_nb,
                int           dim,
                int           k,
                float *       knn_dist,
                int *         knn_index) {

    // Constants
    const unsigned int size_of_float = sizeof(float);
    const unsigned int size_of_int   = sizeof(int);

    // Return variables
    cudaError_t  err0, err1, err2, err3, err4, err5;

    // Check that we have at least one CUDA device
    int nb_devices;
    err0 = cudaGetDeviceCount(&nb_devices);
    if (err0 != cudaSuccess || nb_devices == 0) {
        printf("ERROR: No CUDA device found\n");
        return false;
    }

    // Select the first CUDA device as default
    err0 = cudaSetDevice(0);
    if (err0 != cudaSuccess) {
        printf("ERROR: Cannot set the chosen CUDA device\n");
        return false;
    }

    // Initialize CUBLAS
    cublasInit();

    // Allocate global memory
    float * ref_dev        = NULL;
    float * query_dev      = NULL;
    float * dist_dev       = NULL;
    int   * index_dev      = NULL;
    float * ref_norm_dev   = NULL;
    float * query_norm_dev = NULL;
    size_t  ref_pitch_in_bytes;
    size_t  query_pitch_in_bytes;
    size_t  dist_pitch_in_bytes;
    size_t  index_pitch_in_bytes;
    err0 = cudaMallocPitch((void**)&ref_dev,   &ref_pitch_in_bytes,   ref_nb   *␣
↪size_of_float, dim);
    err1 = cudaMallocPitch((void**)&query_dev, &query_pitch_in_bytes, query_nb *␣
↪size_of_float, dim);
    err2 = cudaMallocPitch((void**)&dist_dev,  &dist_pitch_in_bytes,  query_nb *␣
↪size_of_float, ref_nb);
    err3 = cudaMallocPitch((void**)&index_dev, &index_pitch_in_bytes, query_nb *␣
↪size_of_int,   k);
    err4 = cudaMalloc((void**)&ref_norm_dev,   ref_nb   * size_of_float);
    err5 = cudaMalloc((void**)&query_norm_dev, query_nb * size_of_float);
    if (err0 != cudaSuccess || err1 != cudaSuccess || err2 != cudaSuccess ||␣
↪err3 != cudaSuccess || err4 != cudaSuccess || err5 != cudaSuccess) {
        printf("ERROR: Memory allocation error\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }
    // Deduce pitch values
    size_t ref_pitch   = ref_pitch_in_bytes   / size_of_float;
    size_t query_pitch = query_pitch_in_bytes / size_of_float;
    size_t dist_pitch  = dist_pitch_in_bytes  / size_of_float;
    size_t index_pitch = index_pitch_in_bytes / size_of_int;

    // Check pitch values
    if (query_pitch != dist_pitch || query_pitch != index_pitch) {
        printf("ERROR: Invalid pitch value\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Copy reference and query data from the host to the device
    err0 = cudaMemcpy2D(ref_dev,   ref_pitch_in_bytes,   ref,   ref_nb *␣
↪size_of_float,   ref_nb * size_of_float,   dim, cudaMemcpyHostToDevice);
    err1 = cudaMemcpy2D(query_dev, query_pitch_in_bytes, query, query_nb *␣
↪size_of_float, query_nb * size_of_float, dim, cudaMemcpyHostToDevice);
    if (err0 != cudaSuccess || err1 != cudaSuccess) {
        printf("ERROR: Unable to copy data from host to device\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Compute the squared norm of the reference points
    dim3 block0(256, 1, 1);
    dim3 grid0(ref_nb / 256, 1, 1);
    if (ref_nb % 256 != 0) grid0.x += 1;
    compute_squared_norm<<<grid0, block0>>>(ref_dev, ref_nb, ref_pitch, dim,␣
↪ref_norm_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Compute the squared norm of the query points
    dim3 block1(256, 1, 1);
    dim3 grid1(query_nb / 256, 1, 1);
    if (query_nb % 256 != 0) grid1.x += 1;
    compute_squared_norm<<<grid1, block1>>>(query_dev, query_nb, query_pitch, 
↪dim, query_norm_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Computation of query*transpose(reference)
    cublasSgemm('n', 't', (int)query_pitch, (int)ref_pitch, dim, (float)-2.0, 
↪query_dev, query_pitch, ref_dev, ref_pitch, (float)0.0, dist_dev, query_pitch);
    if (cublasGetError() != CUBLAS_STATUS_SUCCESS) {
        printf("ERROR: Unable to execute cublasSgemm\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Add reference points norm
    dim3 block2(16, 16, 1);
    dim3 grid2(query_nb / 16, ref_nb / 16, 1);
    if (query_nb % 16 != 0) grid2.x += 1;
    if (ref_nb   % 16 != 0) grid2.y += 1;
    add_reference_points_norm<<<grid2, block2>>>(dist_dev, query_nb, dist_pitch,␣
↪ref_nb, ref_norm_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Sort each column
    modified_insertion_sort<<<grid1, block1>>>(dist_dev, dist_pitch, index_dev,␣
↪index_pitch, query_nb, ref_nb, k);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Add query norm and compute the square root of the k first elements
    dim3 block3(16, 16, 1);
    dim3 grid3(query_nb / 16, k / 16, 1);
    if (query_nb % 16 != 0) grid3.x += 1;
    if (k        % 16 != 0) grid3.y += 1;
    add_query_points_norm_and_sqrt<<<grid3, block3>>>(dist_dev, query_nb,␣
↪dist_pitch, k, query_norm_dev);
    if (cudaGetLastError() != cudaSuccess) {
        printf("ERROR: Unable to execute kernel\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }
    // Copy k smallest distances / indexes from the device to the host
    err0 = cudaMemcpy2D(knn_dist,  query_nb * size_of_float, dist_dev, ␣
↪dist_pitch_in_bytes,  query_nb * size_of_float, k, cudaMemcpyDeviceToHost);
    err1 = cudaMemcpy2D(knn_index, query_nb * size_of_int,   index_dev,␣
↪index_pitch_in_bytes, query_nb * size_of_int,   k, cudaMemcpyDeviceToHost);
    if (err0 != cudaSuccess || err1 != cudaSuccess) {
        printf("ERROR: Unable to copy data from device to host\n");
        cudaFree(ref_dev);
        cudaFree(query_dev);
        cudaFree(dist_dev);
        cudaFree(index_dev);
        cudaFree(ref_norm_dev);
        cudaFree(query_norm_dev);
        cublasShutdown();
        return false;
    }

    // Memory clean-up and CUBLAS shutdown
    cudaFree(ref_dev);
    cudaFree(query_dev);
    cudaFree(dist_dev);
    cudaFree(index_dev);
    cudaFree(ref_norm_dev);
    cudaFree(query_norm_dev);
    cublasShutdown();

    return true;
}
```

Here, we define a template to find the most frequent label among the k nearest neighbors to make a prediction:

[0]:
```
template<class T, class U>
struct less_second
{
bool operator()(const std::pair<T, U>& x, const std::pair<T, U>& y)
{
        return x.second < y.second;
    }
    };

    template<class Iterator>
    std::pair<typename std::iterator_traits<Iterator>::value_type, int>
    most_frequent(Iterator begin, Iterator end)
    {
    typedef typename std::iterator_traits<Iterator>::value_type vt;
    std::map<vt, int> frequency;
    for (; begin != end; ++begin) ++frequency[*begin];
    return *std::max_element(frequency.begin(), frequency.end(),
                             less_second<vt, int>());
    }

/*
int array[] = {1,1,2}
std::pair<int, int> result = most_frequent(array, array + 2);
std::cout << result.first << " appears " << result.second << " times.\n";
*/
```

Tests

Here, we test the three implementations of the kNN algorithm defined in the previous section. The output contains the time it takes to compute the distance matrix and sort it.

[0]:
```
#include <algorithm>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>

#include "knncuda.h"



/**
 * Initializes randomly the reference and query points.
 *
 * @param ref        refence points
 * @param ref_nb     number of reference points
 * @param query      query points
 * @param query_nb   number of query points
 * @param dim        dimension of points
 */
void initialize_data(float * ref,
                     int     ref_nb,
                     float * query,
                     int     query_nb,
                     int     dim) {

    // Initialize random number generator
    srand(time(NULL));

    // Generate random reference points
    for (int i=0; i<ref_nb*dim; ++i) {
        ref[i] = 10. * (float)(rand() / (double)RAND_MAX);
    }

    // Generate random query points
    for (int i=0; i<query_nb*dim; ++i) {
        query[i] = 10. * (float)(rand() / (double)RAND_MAX);
    }
}


/**
 * Computes the Euclidean distance between a reference point and a query point.
 *
 * @param ref          refence points
 * @param ref_nb       number of reference points
 * @param query        query points
 * @param query_nb     number of query points
 * @param dim          dimension of points
 * @param ref_index    index to the reference point to consider
 * @param query_index  index to the query point to consider
 * @return computed distance
 */
float compute_distance(const float * ref,
                       int           ref_nb,
                       const float * query,
                       int           query_nb,
                       int           dim,
                       int           ref_index,
                       int           query_index) {
    float sum = 0.f;
    for (int d=0; d<dim; ++d) {
        const float diff = ref[d * ref_nb + ref_index] - query[d * query_nb +␣
 ↪query_index];
        sum += diff * diff;
    }
    return sqrtf(sum);
}


/**
 * Gathers at the beginning of the `dist` array the k smallest values and their
 * respective index (in the initial array) in the `index` array. After this call,
 * only the k-smallest distances are available. All other distances might be␣
 ↪lost.
 *
 * Since we only need to locate the k smallest distances, sorting the entire␣
 ↪array
 * would not be very efficient if k is relatively small. Instead, we perform a
 * simple insertion sort by eventually inserting a given distance in the first
 * k values.
 *
 * @param dist    array containing the `length` distances
 * @param index   array containing the index of the k smallest distances
 * @param length  total number of distances
 * @param k       number of smallest distances to locate
 */
void  modified_insertion_sort(float *dist, int *index, int length, int k){

    // Initialise the first index
    index[0] = 0;

    // Go through all points
    for (int i=1; i<length; ++i) {

        // Store current distance and associated index
        float curr_dist  = dist[i];
        int   curr_index = i;

        // Skip the current value if its index is >= k and if it's higher the␣
 ↪k-th slready sorted mallest value
        if (i >= k && curr_dist >= dist[k-1]) {
            continue;
        }

        // Shift values (and indexes) higher that the current distance to the␣
 ↪right
        int j = std::min(i, k-1);
        while (j > 0 && dist[j-1] > curr_dist) {
            dist[j]  = dist[j-1];
            index[j] = index[j-1];
            --j;
        }

        // Write the current distance and index at their position
        dist[j]  = curr_dist;
        index[j] = curr_index;
    }
}


/*
```

```
 * For each input query point, locates the k-NN (indexes and distances) among␣
 ↪the reference points.
 *
 * @param ref        refence points
 * @param ref_nb     number of reference points
 * @param query      query points
 * @param query_nb   number of query points
 * @param dim        dimension of points
 * @param k          number of neighbors to consider
 * @param knn_dist   output array containing the query_nb x k distances
 * @param knn_index  output array containing the query_nb x k indexes
 */
bool knn_c(const float * ref,
           int           ref_nb,
           const float * query,
           int           query_nb,
           int           dim,
           int           k,
           float *       knn_dist,
           int *         knn_index) {

    // Allocate local array to store all the distances / indexes for a given␣
 ↪query point
    float * dist  = (float *) malloc(ref_nb * sizeof(float));
    int *   index = (int *)   malloc(ref_nb * sizeof(int));

    // Allocation checks
    if (!dist || !index) {
        printf("Memory allocation error\n");
        free(dist);
        free(index);
        return false;
    }

    // Process one query point at the time
    for (int i=0; i<query_nb; ++i) {

        // Compute all distances / indexes
        for (int j=0; j<ref_nb; ++j) {
            dist[j]  = compute_distance(ref, ref_nb, query, query_nb, dim, j, i);
            index[j] = j;
        }

        // Sort distances / indexes
        modified_insertion_sort(dist, index, ref_nb, k);

        // Copy k smallest distances and their associated index
        for (int j=0; j<k; ++j) {
            knn_dist[j * query_nb + i]  = dist[j];
            knn_index[j * query_nb + i] = index[j];
        }
    }

    // Memory clean-up
    free(dist);
    free(index);

    return true;

}



/**
 * Test an input k-NN function implementation by verifying that its output
 * results (distances and corresponding indexes) are similar to the expected
 * results (ground truth).
 *
 * Since the k-NN computation might end-up in slightly different results
 * compared to the expected one depending on the considered implementation,
 * the verification consists in making sure that the accuracy is high enough.
 *
 * The tested function is ran several times in order to have a better estimate
 * of the processing time.
 *
 * @param ref            reference points
 * @param ref_nb         number of reference points
 * @param query          query points
 * @param query_nb       number of query points
 * @param dim            dimension of reference and query points
 * @param k              number of neighbors to consider
 * @param gt_knn_dist    ground truth distances
 * @param gt_knn_index   ground truth indexes
 * @param knn            function to test
 * @param name           name of the function to test (for display purpose)
 * @param nb_iterations  number of iterations
 * return false in case of problem, true otherwise
 */
bool test(const float * ref,
          int           ref_nb,
          const float * query,
          int           query_nb,
          int           dim,
          int           k,
          float *       gt_knn_dist,
           int *          gt_knn_index,
           bool (*knn)(const float *, int, const float *, int, int, int, float *,␣
↪int *),
           const char *   name,
           int            nb_iterations) {

    // Parameters
    const float precision    = 0.001f; // distance error max
    const float min_accuracy = 0.999f; // percentage of correct values required

    // Display k-NN function name
    printf("- %-17s : ", name);

    // Allocate memory for computed k-NN neighbors
    float * test_knn_dist  = (float*) malloc(query_nb * k * sizeof(float));
    int   * test_knn_index = (int*)   malloc(query_nb * k * sizeof(int));

    // Allocation check
    if (!test_knn_dist || !test_knn_index) {
        printf("ALLOCATION ERROR\n");
        free(test_knn_dist);
        free(test_knn_index);
        return false;
    }

    // Start timer
    struct timeval tic;
    gettimeofday(&tic, NULL);

    // Compute k-NN several times
    for (int i=0; i<nb_iterations; ++i) {
        if (!knn(ref, ref_nb, query, query_nb, dim, k, test_knn_dist,␣
↪test_knn_index)) {
            free(test_knn_dist);
            free(test_knn_index);
            return false;
        }
    }

    // Stop timer
    struct timeval toc;
    gettimeofday(&toc, NULL);

    // Elapsed time in ms
    double elapsed_time = toc.tv_sec - tic.tv_sec;
    elapsed_time += (toc.tv_usec - tic.tv_usec) / 1000000.;
    // Verify both precisions and indexes of the k-NN values
    int nb_correct_precisions = 0;
    int nb_correct_indexes    = 0;
    for (int i=0; i<query_nb*k; ++i) {
        if (fabs(test_knn_dist[i] - gt_knn_dist[i]) <= precision) {
            nb_correct_precisions++;
        }
        if (test_knn_index[i] == gt_knn_index[i]) {
            nb_correct_indexes++;
        }
    }

    // Compute accuracy
    float precision_accuracy = nb_correct_precisions / ((float) query_nb * k);
    float index_accuracy     = nb_correct_indexes    / ((float) query_nb * k);

    // Display report
    if (precision_accuracy >= min_accuracy && index_accuracy >= min_accuracy ) {
        printf("PASSED in %8.5f seconds (averaged over %3d iterations)\n",␣
↪elapsed_time / nb_iterations, nb_iterations);
    }
    else {
        printf("FAILED\n");
    }

    // Free memory
    free(test_knn_dist);
    free(test_knn_index);

    return true;
}


/**
 * 1. Create the synthetic data (reference and query points).
 * 2. Compute the ground truth.
 * 3. Test the different implementation of the k-NN algorithm.
 */
int main(void) {

    // Parameters
    const int ref_nb   = 16384;
    const int query_nb = 4096;
    const int dim      = 128;
    const int k        = 16;

    // Display
    printf("PARAMETERS\n");
    printf("- Number reference points : %d\n",   ref_nb);
    printf("- Number query points     : %d\n",   query_nb);
    printf("- Dimension of points     : %d\n",   dim);
    printf("- Number of neighbors     : %d\n\n", k);

    // Sanity check
    if (ref_nb<k) {
        printf("Error: k value is larger that the number of reference points\n");
        return EXIT_FAILURE;
    }

    // Allocate input points and output k-NN distances / indexes
    float * ref        = (float*) malloc(ref_nb   * dim * sizeof(float));
    float * query      = (float*) malloc(query_nb * dim * sizeof(float));
    float * knn_dist   = (float*) malloc(query_nb * k   * sizeof(float));
    int   * knn_index  = (int*)   malloc(query_nb * k   * sizeof(int));

    // Allocation checks
    if (!ref || !query || !knn_dist || !knn_index) {
        printf("Error: Memory allocation error\n");
        free(ref);
            free(query);
            free(knn_dist);
            free(knn_index);
        return EXIT_FAILURE;
    }

    // Initialize reference and query points with random values
    initialize_data(ref, ref_nb, query, query_nb, dim);

    // Compute the ground truth k-NN distances and indexes for each query point
    printf("Ground truth computation in progress...\n\n");
    if (!knn_c(ref, ref_nb, query, query_nb, dim, k, knn_dist, knn_index)) {
        free(ref);
            free(query);
            free(knn_dist);
            free(knn_index);
        return EXIT_FAILURE;
    }

    // Test all k-NN functions
    printf("TESTS\n");
    test(ref, ref_nb, query, query_nb, dim, k, knn_dist, knn_index, &knn_c,     ␣
↪      "knn_c",              2);
    test(ref, ref_nb, query, query_nb, dim, k, knn_dist, knn_index,␣
↪&knn_cuda_global,  "knn_cuda_global",  100);
```

```
    test(ref, ref_nb, query, query_nb, dim, k, knn_dist, knn_index,␣
↪&knn_cuda_texture, "knn_cuda_texture", 100);
    test(ref, ref_nb, query, query_nb, dim, k, knn_dist, knn_index, &knn_cublas,␣
↪      "knn_cublas",       100);

    // Deallocate memory
    free(ref);
    free(query);
    free(knn_dist);
    free(knn_index);

    return EXIT_SUCCESS;
}
```

The code creates a set of reference points randomly initialized, creates a set of query points randomly initialized, computes the ground-truth k-NN using a non-optimized C implementation. For each CUDA implementation listed above, the output (indexes and distances) is compared with the ground-truth and the measurement of the processing time is done.

This code was tested on a TESLA V100 GPU on the Adroit computer cluster at Princeton.

To run the code:

1. Compile the code in the code directory using: $Make

2. Test the code using: $ ./test

Results

PARAMETERS

- Number reference points : 16384
- Number query points : 4096
- Dimension of points : 128
- Number of neighbors : 16

TESTS

- `knn_cuda_global` : passed in 0.04879 seconds (averaged over 100 iterations)
- `knn_cuda_texture` : passed in 0.06305 seconds (averaged over 100 iterations)
- `knn_cublas` : passed in 0.03175 seconds (averaged over 100 iterations)

P.S.: To run the code on multiple GPUs, we need to add a few lines of code to split the input data among the GPUs. I will add it as well as the code to compute the algorithm's prediction accuracy when I apply the algorithm to an actual dataset.

References

[1] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. *CoRR*, abs/0804.1448, 2008.

[2] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. pages 3757–3760, 09 2010.

[3] Qi Li, Vojislav Kecman, and Raied Salman. A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu. pages 208–213, 12 2010.