# Project 3: Integrate Sweet KNN into cuML

CSC 512 Final Project

Benjamin Demick
North Carolina State University
Raleigh, North Carolina, USA
bjdemick@ncsu.edu

## Abstract

This report summarizes the motivation, objectives, challenges, solution, lessons and experiences, results, remaining issues, and possible solutions for the integration of the Sweet KNN reference algorithm with the cuML GPU-based machine learning framework.

## 1 Motivation

The motivation for this project was to become familiar with a cutting-edge optimization technique, and become experienced working with the codebase of a large open-source software project. For this project, a recent improvement to the K-nearest neighbors algorithm, Sweet KNN [2] was to be added to the popular cuML machine learning framework for RapidsAI [8] as an alternative algorithm to the brute force algorithm distributed with the framework. Sweet KNN is an implementation of a triangle inequality-based optimization [4] for the K-nearest neighbors algorithm, optimized for GPUs. K-nearest neighbors is a very popular algorithm for classification and recommender systems, so performance improvements have a great potential for impact where KNN is used.

## 2 Objectives

The objectives for this project were to:

- Add a "sweet" algorithm as an option to the cuML NearestNeighbors Python interface
- Integrate the reference implementation with cuML
- Verify the equivalence of the Sweet KNN algorithm against the brute force algorithm used by cuML
- Test the performance of the Sweet KNN algorithm against the standard brute force algorithm used by cuML
- Optimize the performance of the Sweet KNN algorithm and develop tests to illustrate the performance improvement

A reference implementation for Sweet KNN developed in the CUDA programming language [3] was provided, as well as access to the university's GPU cluster for testing and development.

*CSC 512, December 2019, Raleigh, NC, USA*
2019.

## 3 Challenges

This project provided several challenges that had to be overcome in order to perform the required objectives.

### 3.1 Development Server Configuration

The GPU cluster that was provided was not configured properly for CUDA build systems that utilize a CMake-based build configuration, which is how the cuML project is built. While CMake officially supports the CUDA compiler, nvcc, the CUDA Toolkit must be installed properly such that nvcc is built and configured to find the proper header and library paths without extra environment configuration. Examination of the CMake source code confirmed that the program expects nvcc to compile a dummy program with no extra environment configuration. Official developer support forums for CUDA [7] support this, and the CUDA guide on web page for the university cluster [5] even explicitly shows example Makefile-based build commands that indicate that the system is misconfigured. The administrators of the cluster were unable to fix this issue, so the instructor provided a separate GPU server that was configured properly. However, at least three weeks of research time was lost to this issue.

While the alternative server was configured properly for building CMake-based projects and was good for building cuML, it appeared to be misconfigured for profiling CUDA applications due to a mismatch between the compiler and the profiling tools. Some basic profiling was possible, but performing in-depth analysis using tools like NVIDIA Visual Profiler [11] and Nsight [10] was not possible with this server due to the configuration issue. Also, given the amount of users accessing the server for this project, timing can vary widely across different runs of the program.

### 3.2 cuML Development Environment

The development environment for cuML has many dependencies and is not a straightforward experience to configure, build, and run modifications to the cuML source code [9]. Without administrative access to the host system, the Anaconda Distribution [1] is the best way to manage the large number of dependencies needed for the build environment. Finding the right version of Anaconda that was compatible with cuML also required experimentation.

### 3.3 Sweet KNN Reference Implementation

The CUDA Sweet KNN reference implementation is rife with poor programming practices, making it extremely difficult to follow the code. Dead code, unused variables, poor use of global variables, unused function arguments, formatting, and poor memory management were all impediments to reviewing and comprehending the code. In addition, data and variables are not named in a manner consistent with the algorithm description in the paper, and there are very few comments present in the code to orient anyone reading it in the right direction.

### 3.4 Machine Learning Frameworks

Machine learning frameworks and APIs seem to be poorly documented and the documentation lacking examples sufficiently approachable by a non-expert. A great deal of time was spent experimenting and learning how to format, manipulate, and process data with cuML and numpy, and there is often no real feedback loop from the frameworks to efficiently notify the user when something has gone wrong. Initially not having experience with these frameworks was a significant challenge to overcome, but was a worthwhile journey to gain some practical hands-on experience.

## 4 Solution

### 4.1 Improving the Sweet KNN Reference Implementation

Upon initial review of the Sweet KNN reference implementation, it was discovered that the code was nearly incomprehensible due to poor programming practices.

#### 4.1.1 Unused Variables and Arguments

A primary source of trouble was that argument lists were reused between functions regardless of whether all of the arguments were used in the function, which masked unused variables and memory from being easily spotted. To start removing dead code, it was necessary to compile with all warnings on, and starting with unused function arguments, clean up the prototypes, and then remove unused variables that had been referenced in the prototypes. This process revealed several unused variables and memory allocations that were removed. Global variables also made the code difficult to read and track data dependencies, so these were eliminated in favor of local variables after merging the unneeded common8.h header file into `knnjoin.cu`.

#### 4.1.2 Dead Code Elimination

Unexpectedly, as unused host memory allocations were being removed and the program was tested, the program stopped working due to memory errors on the GPU. Adding the unnecessary allocations back in allowed the program to run properly again. Running the `Valgrind` and `cuda-memcheck` tools to debug memory issues on the host and device respectively did not reveal any issues, but the instability was manually traced to a *for* loop in the function *clusterReps* on line 305 in `knnjoin.cu` (Reference implementation). When the program crashed, the very last *cudaMalloc* call in this loop appeared to be getting a size value that was exceptionally large. As this value was coming from a host memory structure, it is likely that there exists a heap overflow affecting that allocation, causing a large value to be written that is later dereferenced as the size value for the GPU allocation. As more refactoring and dead code elimination was completed, this memory error stopped happening and was never fully debugged. This particular bug had likely been masked in the original code by the presence of many unused heap buffers in between the data that was being corrupted.

```
cudaMalloc((void **)&rep2q_static_dev,
           qrep_nb * sizeof(R2all_static_dev));
check(status,"Malloc rep2qs_static failed\n");
cudaMemcpy(rep2q_static_dev,
           rep2q_static, // Invalid!
           qrep_nb * sizeof(R2all_static_dev),
           cudaMemcpyHostToDevice);
check(status,"Memcpy rep2qs_static failed\n");
cudaMalloc((void **)&rep2s_static_dev,
           srep_nb * sizeof(R2all_static_dev));
check(status,"Malloc rep2qs_static failed\n");
cudaMemcpy(rep2s_static_dev,
           rep2s_static, // Invalid!
           srep_nb * sizeof(R2all_static_dev),
           cudaMemcpyHostToDevice);
check(status,"Memcpy rep2qs_static failed\n");
```

**Figure 1.** Dead cudaMemcpy Calls

#### 4.1.3 Memory Management Analysis

Once all of the dead code was eliminated, memory management was analyzed and found to have further issues, mostly related to not freeing memory that was allocated. There were also some allocations that were not considered dead code due to being utilized in later operations, but upon analysis of the program logic the later operations were unnecessary, allowing for more dead code removal.

As an example, in Figure 1 we see two memory allocations, immediately followed by memcpy calls to seemingly initialize the memory. However, upon closer inspection of the code, neither source buffer has been initialized at this point in the code so these memcpy operations are actually copying invalid data. The *check* functions are logically incorrect, as they do not check the status of the operation that they report to have, which can potentially mask errors depending on where the *status* variable is set. This is just one example of many that illustrates the dead code and poor programming latent in this project. This memory use analysis resulted in 1 fewer

call to *cudaMalloc*, 4 fewer host to device *memcpy* calls, and far more readable code.

#### 4.1.4 Readability

The next step that was taken to improve the KNN reference implementation was to rename variables to be more consistent with the description of the algorithm in the paper. "Sources" was replaced with "targets, and variables that use "rep" were renamed where possible to use "landmark". The code was formatted with clang-format, and the resulting source file is now much easier to read and maintain for future improvements.

#### 4.1.5 Usability

The command line interface of the code was also improved to accept a specific point for which to print the nearest neighbors. This helped improve the workflow when comparing results across versions of the program and against other algorithms. This version of the code is provided as `ImprovedSweetKnn.zip` in the project submission, and hopefully serves as a better starting point for those modifying or maintaining this code in the future.

### 4.2 Python Interface

The first task for this project was to locate the interface to the nearest neighbors implementation. From the cuML repository root, the implementation of the *NearestNeighbors* class can be found in:

`python/cuml/neighbors/nearest_neighbors.pyx`

As of branch-0.11, the developers have added an "algorithm" argument that defaults to "brute", so this named argument was already present and did not need to be implemented. The Sweet KNN algorithm requires the configuration of landmark numbers for the query and target sets, so these parameters were added to the NearestNeighbors.__*init*__ function, and if "sweet" is passed as the algorithm parameter, a check is present to ensure that the `q_landmark` and `t_landmark` arguments are also present. Following the logic of the NearestNeighbors class, we see that the *fit* function is used to process the target points, and the kneighbors function processes the query points and calls *brute_force_knn* to compute the neighbors of the query points. The *brute_force_knn* method is a C++ method imported via Cython using the header file in:

`cpp/include/cuml/neighbors/knn.hpp`

and is implemented in:

`cpp/src/knn/knn.cu`

Both preceding files paths are relative to the cuML repository root. As the *brute_force_knn* function is implemented in CUDA, this provides a convenient integration point for the Sweet KNN reference implementation. The aforementioned `nearest_neighbors.pyx`, `knn.hpp`, and `knn.cu` are the only cuML files that were modified during integration.

### 4.3 Integration

#### 4.3.1 CUDA Integration

The improved `knnjoin.cu` code was further refactored to package the Sweet KNN functionality into a single function interface with a prototype similar to the *brute_force_knn* function found in `knn.cu`. This function and its dependencies were placed directly into cuML's `knn.cu`, and the header file knn.hpp was updated to define the *sweet_knn* function for use within the cuML Cython code.

There was one minor issue encountered during the integration of *sweet_knn* with the cuML CUDA code. The *sweet_knn* code was using a legacy API for the CUBLAS GPU linear algebra library that was incompatible with cuML. This required changing the `cublas.h` header file required by *sweet_knn* to `cublas_v2.h`, and porting the necessary functions in the *sweet_knn* code to use the new API. The official porting guide [6] was followed to implement these changes, and this was also backported to the improved base CUDA implementation.

#### 4.3.2 Cython Integration

The final step in the integration process was to import the new CUDA *sweet_knn* function within Cython, and interface the imported function with the data from cuML. During this part of the development, it was discovered that the brute force algorithm uses CUDA streams, which different than the implementation for `sweet_knn`. The Cython code for NearestNeighbors generates device arrays directly, and copies the query and target data to the device at the Cython level. At this point, it was not desirable to modify the implementation of *sweet_knn* to match this behavior as this would have required a large number of changes in that code.

Instead of undertaking another large refactoring task, the cuML implementation is modified to additionally create host-based arrays for the input data, which are then converted to raw C-type arrays for use within the *sweet_knn* code. This approach maintains the original Sweet KNN practice of explicit memory management. This allowed the integration to take place with minimal overhead, and may actually be faster than the native cuML cuDF DataFrame and stream-based pattern used for the brute force implementation, although extensive testing was not done to verify. At this point, the integrated code was ready to test.

## 5 Results

### 5.1 Comparison with Brute Force Algorithm

A key assumption was that the reference implementation of Sweet KNN was correct, and that upon integration, differences between Sweet KNN's results and the brute force algorithm were due to integration errors. Once integration of the Sweet KNN code was completed, the first step was to verify that both the *sweet* and *brute* algorithms produced the same results. To test this, the resulting distance arrays from

*sweet* and *brute* were compared with the `numpy.equal()` function, which compares two n-dimensional arrays and returns an n-dimensional array result with **True** in the places where the values match, and **False** where they do not match. The resulting array is then checked to determine the number of points that have neighbors that disagree and the index of each of these points.

For the "canonical" *skin* dataset that is frequently referenced in the paper and used in the reference implementation, it was discovered that the Sweet KNN results did not match the brute force results. Furthermore, the number of disagreeing points varied widely depending on the number of landmarks chosen. To determine if this error was due to integration mistakes, refactoring, or the reference implementation, a specific point was chosen such that the number of disagreeing points were significant, but easy to verify manually. For the *skin* dataset, with the number of query and target landmarks set to 150, and k=31, there were 6 points found to have neighbors that disagreed between the Sweet KNN implementation and the cuML brute force algorithm.

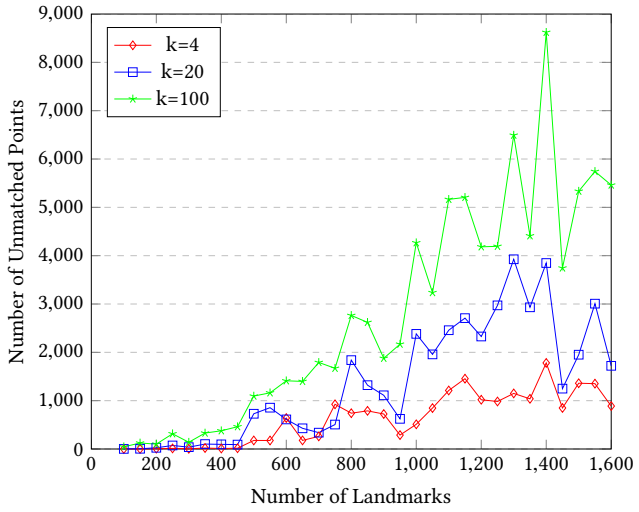


**Figure 2.** Sweet KNN vs. cuML Brute Force KNN (245057 points)

| n | s_index | s_dist | b_index | b_dist |
|---|---|---|---|---|
| 1 | 208764 | 0.00000 | 208764 | 0.00000 |
| 2 | 170168 | 3.00000 | 170168 | 3.00000 |
| 3 | 178056 | 3.00000 | 178056 | 3.00000 |
| 4 | 188124 | 5.65685 | 208765 | 3.16227 |
| 5 | 178055 | 5.65685 | 183646 | 4.58257 |
| 6 | 199856 | 7.34846 | 182540 | 5.38516 |
| 7 | 209324 | 7.81024 | 178055 | 5.65685 |
| 8 | 169609 | 7.81024 | 188124 | 5.65685 |
| 9 | 228661 | 7.87400 | 178061 | 6.48074 |
| 10 | 229220 | 8.12403 | 199856 | 7.34846 |
| 11 | 206527 | 8.30662 | 209324 | 7.81024 |
| 12 | 200974 | 8.30662 | 169609 | 7.81024 |
| 13 | 200415 | 8.60232 | 228661 | 7.87400 |
| 14 | 207086 | 8.66025 | 229220 | 8.12403 |
| 15 | 184199 | 8.66025 | 200974 | 8.30662 |
| 16 | 185339 | 8.83176 | 206527 | 8.30662 |
| 17 | 178054 | 9.00000 | 200415 | 8.60232 |
| 18 | 201533 | 9.00000 | 184199 | 8.66025 |
| 19 | 202651 | 9.00000 | 207086 | 8.66025 |
| 20 | 202092 | 9.00000 | 197026 | 8.71779 |
| 21 | 100524 | 9.27361 | 185339 | 8.83176 |
| 22 | 229219 | 9.48683 | 178054 | 9.00000 |
| 23 | 208782 | 9.64365 | 202651 | 9.00000 |
| 24 | 208778 | 9.64365 | 202092 | 9.00000 |
| 25 | 131752 | 9.64365 | 201533 | 9.00000 |
| 26 | 208796 | 9.69536 | 178616 | 9.11043 |
| 27 | 208780 | 9.89949 | 100524 | 9.27361 |
| 28 | 208766 | 9.89949 | 229219 | 9.48683 |
| 29 | 208776 | 9.89949 | 208778 | 9.64365 |
| 30 | 204328 | 9.89949 | 131752 | 9.64365 |
| 31 | 204887 | 9.89949 | 208782 | 9.64365 |

**Table 1.** Single Unmatching Point Between Sweet and Brute KNN, Landmarks=150, k=31, 245057 points

The k-neighbors results for these 6 points were manually compared to the results from the reference `knnjoin.cu` implementation with the same query and target landmarks and k values, and were found to be in agreement. This evidence strongly suggests that there are bugs in the reference implementation, which are carried through to the integrated version of Sweet KNN. A single point sampling from a disagreeing dataset is shown in Table 1. As can be observed in the table, the distances are close and the Sweet KNN and brute force sets contain many matching values, but the sets are not the same.

To further examine this anomaly, SweetKNN was run with varying landmark value settings, and compared to the brute force algorithm results with the same k value. These results are displayed in Figure 2, and Python scripts are provided in the project submission to facilitate reproduction of these results. The shape of the graphs is roughly the same for different values of k, suggesting that the error may be closely related to computations involving the landmark values.

It was also observed that runtime varied with landmark values. While the brute force algorithm maintained roughly the same runtime, (between 1.6 and 1.8 seconds) regardless of the value of k, the Sweet KNN algorithm was often slower than brute force for larger values of k, and varied with the landmark value setting. These runtime observations are shown in Figure 3.

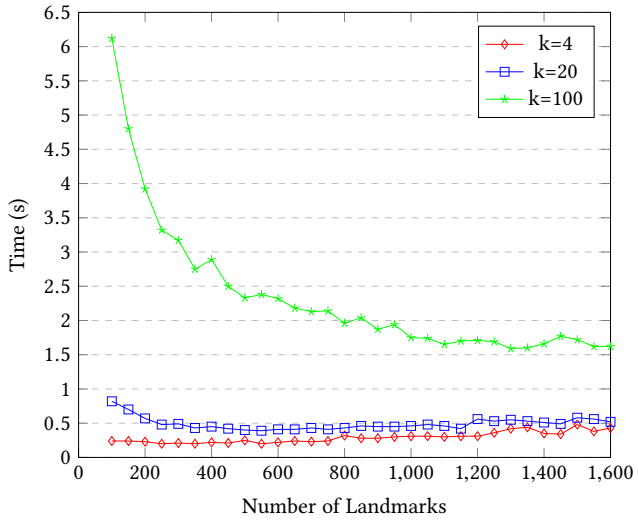


**Figure 3.** Sweet KNN Runtime (245057 points)

### 5.2 Dataset Testing

Six out of the nine datasets referenced in the Sweet KNN paper were tested, with half of the tested datasets causing the program to crash, and the other half completing but providing results that did not agree with the brute force KNN algorithm, as referenced in Section 4.1. It is therefore not valid to perform further performance testing against the brute force algorithm without first determining the root cause of the neighbor calculation errors and fixing the memory corruption errors (which may or may not be related). The status of all dataset testing is shown in Table 2.

| Dataset | Tested | Status |
|---|---|---|
| 3DNet | Y | completes |
| kegg | Y | crash |
| keggD | Y | completes |
| ipums | Y | crash |
| skin | Y | completes |
| arcene | N | N/A |
| kdd | N | N/A |
| dor | N | N/A |
| blog | Y | crash |

**Table 2.** Dataset Status

The untested datasets either contained label information or were difficult to properly process for testing. For detailed discussion on the crashing issue, see Section 7.2.

## 6 Lessons Learned

Debugging issues between the host and the device can be cumbersone, especially if overflows occur on the host that impact values used in memory allocation on the GPU. cuda-gdb combined with the -G option to enable debug information in the executable were helpful in locating the exact source of the aforementioned memory issue, but it is a tenuous operation to debug when a possible dataflow crosses boundaries between the host and the GPU.

Validating results with ML tools isn't particularly straightforward, and there are a lot of published examples on the Internet that are simply incorrect. For instance, as the Sweet KNN algorithm was returning results that seemed legitimate but were different from the cuML brute force implementation, an attempt to verify that the brute force algorithm in cuML also got the same answers as the KNN algorithm in scikit learn to ensure that brute the brute force method was a valid source of ground truth and was not the victim of an implementation error. Following a reference in the online documentation for cuML to compare results to scikit learn simply hung, and after an hour the process was terminated.

The cuML implementation can make use of cuDF DataFrames or numpy ndarray objects as containers for the input data. However, it was observed that cuDF DataFrames, while easier to perform preprocessing on, incur a significant overhead. For the *skin* dataset, it was observed that using an ndarray structure resulted in a total speedup of nearly 7x. It would be interesting to perform more experiments with more datasets to determine what the average overhead is and examine if there are any opportunities to speed up the cuDF implementation.

## 7 Discussion

### 7.1 Major Bugs

There appear to be at least two serious issues that occur in both the reference knnjoin.cu and the improved knnjoin.cu code that prevent further exploration:

1. There is at least one, if not several serious memory corruption bugs in the code. (See Section 7.2)
2. For datasets that did complete, their results do not agree with the results from the brute force implementation.

The memory corruption issues should be corrected first, as it is possible that these are responsible for the variance in the results. Once the memory corruption root cause is identified and addressed (this should be done with the CUDA implementation), the corrected code should be re-integrated with cuML and retested against the brute force algorithm to determine if the memory corruption issue was also the root cause of the result disagreement between the two algorithms. The improved and refactored CUDA implementation should be a good starting point to correct these issues, and there are clear organizational comments within the supplied code to facilitate easy porting of any changes to the CUDA implementation into the cuML project. This must be completed before any algorithm performance experimentation and evaluation is done.

```
__device__ float Edistance_128(
        float *a, float *b, int dim) {
  float distance = 0.0f;
  float4 *A = (float4 *)a;
  float4 *B = (float4 *)b;
  float tmp = 0.0f;
  for (int i = 0; i < int(dim / 4); i++) {
    // Misaligned access at occurs at A[i]
    float4 a_local = A[i];
    float4 b_local = __ldg(&B[i]);
    tmp = a_local.x - b_local.x;
    distance += tmp * tmp;
    tmp = a_local.y - b_local.y;
    distance += tmp * tmp;
    tmp = a_local.z - b_local.z;
    distance += tmp * tmp;
    tmp = a_local.w - b_local.w;
    distance += tmp * tmp;
  }
  for (int i = int(dim / 4) * 4; i < dim; i++) {
    tmp = (a[i]) - (b[i]);
    distance += tmp * tmp;
  }
  return sqrt(distance);
}
```

**Figure 4.** Function where Crash Occurs

### 7.2 Memory Corruption Issues

All three of the datasets that cause crashes appear to do so at the same location in the *sweet_knn* code. This occurs in every implementation (reference, improved, and integrated). One of the crashing datasets (*ipums*) was reformatted to work with the reference and improved versions of knnjoin.cu to allow for finer-grained control over the debug process.

Executing knnjoin with cuda-gdb and continuing until an exception occurs indicates a Warp Misaligned Address exception, occurring in the `Edistance_128` in Figure 4 function, which is called by `selectReps_cuda`.

The root cause of this memory corruption has not been found at the time of this report. Instructions for reproducing this error with the base CUDA code are documented and included with the project submission.

### 7.3 More Code Improvements

There are still many improvements that can be made to the readability and maintainability of the Sweet KNN code. Production code should use standard logging interfaces rather than calls to *printf*, and statistics where necessary should be passed back up to the calling functions.

There are also still likely hidden memory errors due to poor memory management practices. For example, there are still several locations in the code where a memory allocation routine is called, but no freeing of this memory is ever done. Comments should be added where appropriate to related functions back to the published paper. This would allow for easier cross-referencing and validation of the algorithm.

The code currently does not handle errors at all. The original developers simply report on (some) errors when detected, but continue on with the rest of the program with no regards to resource cleanup or error handling. This should be robustly implemented and properly integrated with cuML error handling practices prior to any pull request to cuML.

### 7.4 Integration Point

It is possible to integrate the Sweet KNN CUDA code within `cuml/src/src_prims/selection/knn.h`, and this may be more consistent with development practices for the project. The currently-implemented integration required a lower number of source files to be changed, and given the other challenges, a simpler integration point was chosen. An improvement that may be made prior to submitting a pull request to cuML would be to consult with the cuML development team directly to determine the best point of integration.

### 7.5 GPU Performance Evaluation

Due to the myriad issues that were encountered throughout this project, it was not possible to objectively evaluate the performance of the GPU-based code. As this was a primary objective for this project, it is disappointing that it was impossible to realize.

## 8 Conclusion

While there were considerable challenges encountered during the implementation of this project, there are several notable successes, accomplishments, and experiments that were performed. Each of these contribute to the knowledge of the Sweet KNN implementation and its ability to be integrated with cuML once the implementation bugs are fixed.

- The reference code was successfully and efficiently integrated with cuML
- Several implementation bugs were identified and analyzed
- The existing reference implementation was considerably refactored to remove dead code, improve readability, improve usability, and improve performance
- Several tests were written and submitted that can be used to reproduce results and assist with debugging and validation
- Many new tools were learned, especially from the CUDA toolset and machine learning libraries

Overall, the project was a successful venture with a net positive gain for the Sweet KNN implementation.

## References


[1] Anaconda, Inc. [n. d.]. Anaconda Distribution. https://www.anaconda.com/. Accessed December 11, 2019.

[2] Guoyang Chen, Yufei Ding, and Xipeng Shen. 2017. Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 621–632.

[3] Chen, Guoyang. [n. d.]. Very Fast KNN implementation on GPU with Triangle Inequality Theory. https://people.engr.ncsu.edu/xshen5/csc512_fall2019/knn_sweet-master.zip. Accessed December 17, 2019.

[4] Yufei Ding, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Top: A framework for enabling algorithmic optimizations for distance-related problems. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1046–1057.

[5] North Carolina State University. [n. d.]. ARC: A Root Cluster for Research into Scalable Computer Systems. https://arcb.csc.ncsu.edu/~mueller/cluster/arc/. Accessed December 11, 2019.

[6] nVidia. [n. d.]. CUDA Toolkit 4.2 CUBLAS Library. https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf. Accessed December 16, 2019.

[7] nVidia. [n. d.]. CUDA ZONE Forums, Accelerated Computing, CUDA Setup and Installation. https://devtalk.nvidia.com/default/topic/1061159/cc1plus-fatal-error-cuda_runtime-h-no-such-file-or-directory-compilation-terminated/. Accessed December 11, 2019.

[8] nVidia. [n. d.]. cuML - RAPIDS Machine Learning Library. https://github.com/rapidsai/cuml. Accessed December 11, 2019.

[9] nVidia. [n. d.]. cuML Build From Source Guide. https://github.com/rapidsai/cuml/blob/branch-0.11/BUILD.md. Accessed December 11, 2019.

[10] nVidia. [n. d.]. Nsight Eclipse Edition. https://developer.nvidia.com/nsight-eclipse-edition. Accessed December 17, 2019.

[11] nVidia. [n. d.]. nVidia Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. Accessed December 17, 2019.