GP-GPU Project Report

Zakaria El Kazdam and Riham Faraj

February 16, 2025

1 Introduction

Locality-Sensitive Hashing (LSH) is a technique for approximate nearest neighbor search in high-dimensional spaces. The goal is to map similar points into the same hash buckets with high probability, allowing efficient retrieval.
Initially, a sequential implementation of LSH was developed to validate correctness and measure computational cost. However, as data size and the number of hash functions grow, the sequential approach becomes inefficient due to its computational bottlenecks.
To address this, a parallel version using CUDA is introduced to leverage GPU parallelism. The key motivations for parallelization include:

- **Massive speedup**: CUDA enables thousands of threads to compute hash functions simultaneously.
- **Optimized memory usage**: GPU memory architecture accelerates large-scale computations.
- **Scalability**: The parallel approach efficiently handles larger datasets compared to the sequential version.

This report details the parallel LSH implementation, including hash function generation, parallel computation, and GPU memory management.

2 Sequential implementation

The goal of the implementation is to hash and access similar data points in less computational effort of the nearest neighbor search. The system is sequential and divided into multiple components for modularity and reusability.

2.1 Objectives

- Develop an efficient LSH algorithm for nearest neighbor searching.
- Divide data points based on randomized hashing.
- Provide a sequential implementation for easy integration and testing.

2.2 Architecture

The work is divided into header files (.h) and source files (.cpp), well organized and modulated.

2.2.1 Main Components

File	Description
DLSH_algo.cpp / DLSH_algo.h	Holds the source code of the basic LSH algorithm, like hash table establishment and search.
hashing.cpp / hashing.h	Defines hash operations and mathematical function requirements for hashing.
NewPoint.cpp / NewPoint.h	Operates on new data points and their hashed variants and look up.
make_csv.cpp	Converts .npy (NumPy) files to CSV for easy data handling.

2.2.2 Data Flow

1. **Preprocessing**: Reads data from CSV files (or.npy format) and structure it for hashing.
2. **Hashing**: Maps data points onto random hyperplanes through hash functions.
3. **Storage**: Points are assigned to buckets in multiple hash tables.
4. **Search & Retrieval**: Given a query point, the system looks for similar points in the same hash buckets.

2.3 Core Functionalities

2.3.1 Locality-Sensitive Hashing (LSH)

- Embeds data points in lower-dimension space employing randomly drawn hash functions.
- Assigns data points to hash buckets based on vector projections.
- Reduces the complexity of nearest-neighbor search from $O(n)$ to sublinear time.

2.3.2 Hash Function Definition

Each hash function $h(x)$ is defined as:

$$h(x) = \frac{a \cdot x + b}{w} \tag{1}$$

where:

- a - A random projection vector.
- b - A uniformly random bias.
- w - A user-defined bucket width.

2.3.3 Multi-Level Hashing Strategy

- **First Level (L_1)**: Assigns points to broad categories.
- **Second Level (L_2)**: Further refines lookup accuracy.
- Uses multiple hash tables to increase retrieval probability.

2.3.4 Efficient Data Storage

- The system uses nested maps (`std::map`) and sets (`std::set`) to store hashed values.
- Implements a custom vector comparator to handle multi-dimensional data in hash maps.

2.4 Performance Analysis

2.4.1 Computational Complexity

Operation	Complexity
Hash Computation	$O(d)$ (for d-dimensional data)
Insertion into Hash Table	$O(1)$
Nearest Neighbor Search	$O(L_1 + L_2)$ (depends on number of hash tables)

2.4.2 Space Complexity

- Uses multiple hash tables, which trade space for faster lookups.
- Hash storage depends on:
  - Number of hash tables (L_1, L_2).
  - Data dimensionality (n).
  - Bucket width (w).

2.5 Potential Improvements

- **Parallelization**: Can use CUDA to accelerate hashing and retrieval.
- **Adaptive Hashing**: Dynamic w1, w2, L1, and L2 selection for improved accuracy.

3 Parallel Implementation

3.1 Hashing functions structure

When transitioning from the sequential to the parallel implementation of LSH, the first major modification was in the representation of the hashing function structure ('HashingFunct').
In the sequential implementation, the 'HashingFunct' structure used `std::vector<double> a` , which is convenient for CPU-based execution but inefficient for parallelism because CUDA kernels cannot directly access 'std::vector', and **STL functions and methods are not available in CUDA device code, as CUDA does not support the full C++ standard library on the device.**
To enable efficient GPU computation, the structure was modified to use a raw pointer `double* a`, pointing to GPU memory. This change eliminates unnecessary memory transfers, allowing CUDA threads to directly access 'a' in GPU memory, improving performance and scalability.

3.2 CUDA Kernel: `generateLSHParams`

3.2.1 Main Idea

The `generateLSHParams` kernel is responsible for initializing the **hashing function parameters** in parallel using CUDA. Each **hash function** in LSH is defined by:

- A **random vector** a, whose coordinates are drawn from a **normal distribution**.
- A **random bias** b, drawn from a **uniform distribution** in $[0, w]$.

Since LSH requires multiple independent hash functions, this kernel ensures that each function receives **unique random parameters** efficiently. Instead of generating them sequentially on the CPU, **CUDA parallelizes** this process, allowing each thread to compute parts of a in parallel.

3.2.2 Kernel Parameters Explanation

```
__global__ void generateLSHParams(double* d_a, double* d_b, int n,
                                  double w, unsigned long long seed, int funcIndex)
```

- `double* d_a`: Stores all **random vectors** a for all hash functions.
  - **Memory layout:** The vectors are **concatenated** sequentially in GPU memory.
  - The index **offset = funcIndex * n** ensures each function writes to its correct position.

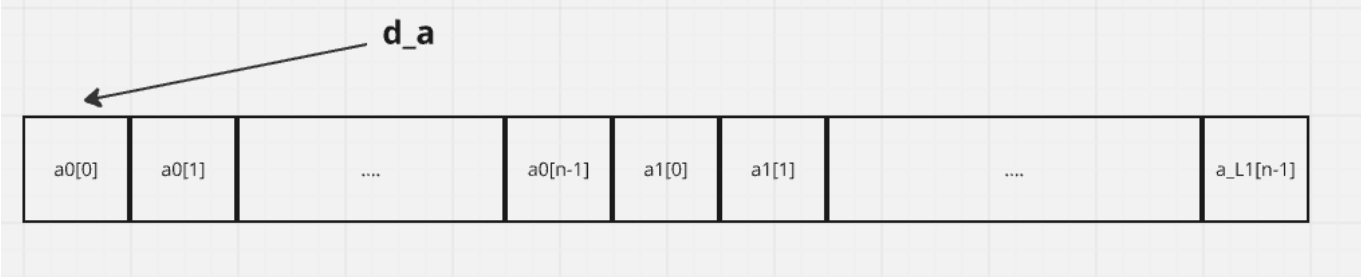

Figure 1: Representation of the organization of the a vectors of the hashing functions in `d_a`

- `double* d_b`: Stores the **bias values** b for each hash function.
  - Only **one thread** (`i == 0`) generates b per hash function.
- `int n`: The **dimensionality** of the hash function (i.e., the size of a).
- `double w`: The **bin width** which controls the scaling of b.
  - Determines the bucket width in the LSH hashing process.
- `unsigned long long seed`: The **random seed** for `curand_init()`.
  - Ensures different executions generate different random numbers.
- `int funcIndex`: The **index of the hash function being processed**
  - Used to calculate **offsets** in memory so each function stores its values correctly.

3.2.3 Parallel Execution and Thread Mapping

- **Each thread computes one coordinate** of a, ensuring all n elements are generated in parallel.
- **Only thread** $i = 0$ **computes** b, avoiding redundant calculations.
- **Different hash functions** are initialized independently using `funcIndex`.

This approach **maximizes parallelism** efficiently distributing the workload among CUDA threads while ensuring each hash function receives unique random parameters.

3.3 `hashingComputingCUDA`

3.3.1 Function Description

The hashingComputingCUDA function computes the hash value of a given data point using an LSH hash function. It takes as input a data point, applies the function's parameters, and returns the corresponding hash bucket. This computation is performed entirely on the GPU for efficiency.

```
__device__ int hashingComputingCUDA(const double* d_point, const HashingFunct h, int n)
```

3.3.2 Understanding device

The device qualifier specifies that this function executes on the GPU and can only be called from another GPU function (e.g., a CUDA kernel or another device function).

3.3.3 Parallel Dot Product and Hash Computation – Previous Approach

Initially, a **parallelized version of the dot product computation** was implemented inside `hashingComputingCUDA`, distributing the workload across multiple CUDA threads.

- First Kernel (multiplyElements)

```
37  __global__ void multiplyElements(double* d_point, double* d_a, double* d_partialSum, int n) {
38      __shared__ double sharedMem[256]; // Shared memory for block-level reduction
39      int i = threadIdx.x + blockIdx.x * blockDim.x;
40      int tid = threadIdx.x;
41
42      double product = (i < n) ? d_point[i] * d_a[i] : 0.0;
43      sharedMem[tid] = product;
44      __syncthreads();
45
46      for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
47          if (tid < stride) {
48              sharedMem[tid] += sharedMem[tid + stride];
49          }
50          __syncthreads();
51      }
52
53      if (tid == 0) {
54          d_partialSum[blockIdx.x] = sharedMem[0]; //d_partial sum contains the sum of every block stored on it's index in the vector
55      }
56  }
57
```

Figure 2: code of the first kernel : multiplyElements

Each thread computes a single multiplication between a coordinate of the input vector and the corresponding coordinate in a. The partial results are stored in shared memory to allow efficient block-wise reduction. A parallel reduction sums the elements in shared memory, progressively reducing the number of active threads. The final sum of each block is stored in `d_partialSum`.

- Second Kernel (sumPartialSums) The partial sums from all blocks are summed again using

```
59  __global__ void sumPartialSums(double* d_partialSum, double* d_finalSum, int numBlocks) {
60      __shared__ double sharedMem[256];
61      int tid = threadIdx.x;
62
63      if (tid < numBlocks) {
64          sharedMem[tid] = d_partialSum[tid];
65      } else {
66          sharedMem[tid] = 0.0;
67      }
68      __syncthreads();
69
70      for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
71          if (tid < stride) {
72              sharedMem[tid] += sharedMem[tid + stride];
73          }
74          __syncthreads();
75      }
76
77      if (tid == 0) {
78          *d_finalSum = sharedMem[0];
79      }
80  }
```

Figure 3: code of second kernel :sumPartialSums

another parallel reduction, obtaining the final dot product value. The final result is normalized using b and w, then converted to an integer hash value.

However, this approach **conflicted with the existing structure** of `computeHashes`, causing **memory and execution inconsistencies**

- The function `computeHashes` **already launches one thread per data point**, meaning that **each thread is responsible for computing a full hash** for a single data point.
- However, the **parallel dot product implementation expected multiple threads to collaborate** on computing the dot product of a single data point.
- This caused a **threading mismatch**—each thread launched by `computeHashes` was supposed to compute a full hash independently, but inside `hashingComputingCUDA`, the workload was split across multiple threads.

3.3.4 The Final Decision

Given these conflicts, we **abandoned the parallelization of the dot product** and instead opted for a **sequential computation inside each thread** of `computeHashes`. This ensures:

- **Each thread fully computes its assigned hash independently**, avoiding synchronization issues.
- **No extra memory conflicts**, as each thread operates on its own portion of the data.
- **Better efficiency in LSH scenarios**, since the number of dimensions (`n`) is typically not large enough to justify complex inter-thread reductions.

By simplifying `hashingComputingCUDA` to a **single-threaded sequential dot product**, we eliminated the structural conflicts and ensured correctness in the parallel execution of `computeHashes`.

3.4 CUDA Kernel: `computeHashes`

3.3.1 Kernel Function

The computeHashes kernel is responsible for computing LSH hash values for all data points in parallel. Each thread processes one data point, computing its corresponding L1 and L2 hash values.

```
__global__ void computeHashes(double* d_points, int* d_hash1, int* d_hash2,
                              HashingFunct* d_hashfunctions1, HashingFunct* d_hashfunctions2,
                              int num_points, int L1, int L2, int D) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < num_points) {
        for (int j = 0; j < L1; j++) {
            d_hash1[i * L1 + j] = hashingComputingCUDA(d_points + i * D, d_hashfunctions1[j], D);
        }
        for (int j = 0; j < L2; j++) {
            d_hash2[i * L2 + j] = hashingComputingCUDA(d_points + i * D, d_hashfunctions2[j], D);
        }
    }
}
```

Figure 4: code du troisième kernel

3.3.2 Explanation of `dhash1` and `dhash2`

The arrays `dhash1` and `dhash2` store the computed hash values for each data point. These arrays are indexed such that each point has multiple hash values corresponding to different hash functions.

- `dhash1`: Stores the LSH hash values computed using the first set of hash functions (`dhashfunctions1`).
- `dhash2`: Stores the LSH hash values computed using the second set of hash functions (`dhashfunctions2`).

3.4.3 How These Arrays Are Filled

Each thread is assigned to process a single data point. For each data point i, the thread iterates over: L1 hash functions (to fill `dhash1`). The computed hash values are stored using an offset-based indexing: dhash1[i * L1 + j] stores the j-th hash for point i.
Since each thread computes an entire hash vector for one data point, this structure ensures that hash values are stored efficiently without inter-thread synchronization issues.

3.5 Host Function: `generateLSHParamsOnGPU`

```
std::pair<std::vector<HashingFunct>, double*> generateLSHParamsOnGPU(int n, double w, int numFunctions) {
    double *d_a, *d_b;

    cudaMalloc(&d_a, numFunctions * n * sizeof(double));
    cudaMalloc(&d_b, numFunctions * sizeof(double));

    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    for (int funcIndex = 0; funcIndex < numFunctions; funcIndex++) {
        generateLSHParams<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, n, w, time(NULL) + funcIndex * 1234, funcIndex);
        cudaDeviceSynchronize();
    }

    std::vector<double> h_b(numFunctions);
    cudaMemcpy(h_b.data(), d_b, numFunctions * sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(d_b);
    std::vector<HashingFunct> hashFunctions(numFunctions);
    for (int i = 0; i < numFunctions; i++) {
        hashFunctions[i].a = d_a + i * n;
        hashFunctions[i].b = h_b[i];
        hashFunctions[i].w = w;
    }
    return {hashFunctions, d_a};
}
```

Figure 5: code of generateLSHParamsOnGPU

3.5.1 Function Overview

The generateLSHParamsOnGPU function is responsible for allocating memory on the GPU, generating hash function parameters in parallel, and transferring necessary data back to the host. It returns a pair containing:launched
A **vector of HashingFunct structures**, each representing a hashing function and a **device pointer to d_a**, which stores all generated random vectors in GPU memory.

```
std::pair<std::vector<HashingFunct>, double*> generateLSHParamsOnGPU
```

3.5.2 Step-by-Step Breakdown

- Grid Configuration Calculation

  ```
  int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
  ```

  Instead of using **ceil(n / threadsPerBlock)**, we use this formula because:

  CUDA does not provide a built-in ceil() function for integer division. Integer division truncates decimals, so using (n + threadsPerBlock - 1) / threadsPerBlock ensures proper rounding up without requiring floating-point operations. This method avoids unnecessary computations and improves performance while guaranteeing that even if n is not perfectly divisible by threadsPerBlock, an additional block is launched to handle the remaining threads.

- Why The Copy Loop?

  ```
  std::vector<HashingFunct> hashFunctions(numFunctions);

  for (int i = 0; i < numFunctions; i++) {
  ```

```
hashFunctions[i].a = d_a + i * n;

hashFunctions[i].b = h_b[i];

hashFunctions[i].w = w; } ;
```

This loop assigns each hash function its respective parameters without duplicating data.

hashFunctions[i].a = d_a + i * n; Uses pointer arithmetic to assign each function a different segment of d_a. **Avoids extra memory allocation, as a remains in GPU memory.**

hashFunctions[i].b = h_b[i]; Since b values were already copied to the host, we simply assign them.

hashFunctions[i].w = w; The width parameter is the same for all hash functions.

Thus, this loop ensures minimal memory overhead while efficiently organizing hash functions.

- Why Not Free d_a Like d_b, and Why Return It?

d_b is freed because we copied its values back to the host. d_a remains allocated because it stores the vectors of all hash functions. **If we freed d_a, every hash function's a would point to invalid memory, leading to segmentation faults.**
We return d_a to ensure it remains accessible for future GPU computations (e.g., computeHashes). The caller of generateLSHParamsOnGPU is responsible for freeing d_a when it's no longer needed.

3.6 Host Function: finalHashCUDA

The finalHashCUDA function constructs the final hash table by computing LSH hash values for each data point and organizing them into a nested unordered_map. Each point is assigned two hash vectors (hash1 and hash2), which serve as keys for hierarchical indexing. The function stores points in corresponding hash buckets, enabling efficient retrieval of approximate nearest neighbors. Since hashingComputingCUDA runs on the GPU, this approach ensures fast hash computations

3.7 `Main Function`

The main function serves as the entry point of the LSH implementation, **integrating all key components into a complete pipeline**. It first loads the dataset, then generates LSH hash functions in parallel using CUDA, ensuring efficient parameter initialization. After that, it constructs the hash table by computing and storing LSH hash values for all data points. To demonstrate the functionality, it generates a random point, computes its hash values, and searches for neighbors within the same hash bucket. This structured execution ensures that the parallelized hashing approach is both efficient and scalable, allowing for fast approximate nearest neighbor search.

4 Future Improvements and Potential Enhancements

Due to time constraints caused by **unforeseen circumstances** (Zakaria's fractured hand and the time spent in the hospital from January 11 until today), we focused on implementing the core LSH functionality while ensuring efficient parallel execution. However, several **potential enhancements** could further optimize and expand this project. One of the key improvements we initially considered was **parallelizing the scalar product computation in hashingComputingCUDA** while maintaining compatibility with computeHashes. Although we faced synchronization challenges, an optimized approach using **warp-level reductions or shared memory optimizations** could significantly enhance performance. Additionally, while we carefully handled memory allocation and deallocation, integrating **CUDA memory debugging tools such as cuda-memcheck** would provide deeper insights into potential leaks and inefficiencies, ensuring perfect memory management.