MASTER THESIS

Drahomír Hanák

Meeting the challenges of k-nearest neighbour search implementation for GPU accelerators

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Kruliš Martin, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2023

I would like to thank my supervisor, doc. RNDr. Kruliš Martin, Ph.D., for his invaluable advice, constructive criticism, and the time he dedicated to helping me write this thesis.

Title: Meeting the challenges of k-nearest neighbour search implementation for GPU accelerators

Author: Drahomír Hanák

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Kruliš Martin, Ph.D., Department of Distributed and Dependable Systems

Abstract: Similarity search is a commonly used technique in databases for finding objects similar to a query. It finds applications in content-based retrieval of complex objects like images, information retrieval, and statistical learning. Our thesis focuses on the implementation and optimization of the k nearest neighbours (kNN) algorithm on a GPU, a commonly used technique in similarity search. We analyze and evaluate several existing GPU kNN implementations in various configurations and propose the best algorithm for each configuration. We also suggest optimizations of k-selection. In particular, we suggest a small k-selection approach, which achieves up to 80% of peak theoretical throughput on a typical configuration used in many applications of kNN and is faster than the current state-of-the-art. We implemented a fused algorithm, which solves kNN without materializing the distance matrix, and a large k-selection, which outperforms an optimized, parallel sorting of the whole database by a significant margin.

Keywords: kNN top-k parallel GPU CUDA

Contents

1. Introduction

Similarity search is a general technique used in databases to find objects similar to a query based on a similarity between pairs of objects. It is often employed in large-scale databases of complex objects such as images to facilitate content-based retrieval. Similarity search finds applications in many other fields, including information retrieval, statistical learning, and data visualization, among others. For example, web search engines use similarity search to rank web pages based on their similarity to a query.

The k nearest neighbour (kNN) algorithm is one technique often used in similarity search. Given a database of objects, a query object and a similarity function, we want to find the k closest objects to the query according to the similarity function. While a serial implementation of this algorithm is quite simple, the dataset size might render this approach entirely infeasible. Hence, better implementations, which can utilize the parallelism of modern hardware, are required. In particular, many-core architectures, such as graphics processing units (GPUs), have been used for this purpose [29, 26, 22].

GPUs were originally intended for processing graphics where it is necessary to compute values of individual vertices of 3D models or pixels in a massively parallel fashion so that the whole frame composed of millions of pixels is processed in order of milliseconds. GPUs have thus been optimized for throughput rather than the latency of individual operations. The programming model of GPUs has been extended since their first introduction so that they can be used for general-purpose computations.

The problem of kNN is, at least in part, well suited for GPUs since distance computations can be solved using a data-parallel approach which benefits from GPU architecture. However, an efficient implementation of kNN on modern GPUs is non-trivial and requires utilizing many aspects of the hardware architecture. Moreover, GPU architecture is dissimilar to its CPU counterpart, so specialized algorithms have to be developed to take full advantage of hardware resources.

A large body of work has been done on this topic. However, an optimal implementation depends on the configuration of the problem. The size of the database and the magnitude of k, for instance, both have a substantial impact on performance and can vary widely depending on the domain of the application. To the best of our knowledge, no publication has done a complex performance overview of the problem for various configurations.

The contribution of this thesis is twofold:

- We evaluate existing GPU kNN implementations for various configurations of the problem and propose the best method for each tested configuration.

- We analyze GPU implementations of kNN and propose optimizations for several parts of the problem. We show that an algorithm proposed by Kruliš et al. [29] can be modified using ideas from Tang et al. [44] to perform better than the current state-of-the-art for small k. We also present a fused kernel based on the work of Kruliš et al. [30], which solves the kNN problem without materializing the distance matrix. This approach is not only faster than the previously mentioned method in some instances, but it also has a lower memory footprint. The latter point is especially important on GPUs which have a very limited amount of available memory.

1.1 Taxonomy

There are several variants of the kNN problem [36]. *Structureless* solutions compute all distances between query and database points. The result is then used to find the k closest objects. Another option is to create an indexing structure. Indexing structures usually utilize properties of the distance function to compute the lower bound of the actual distance so that some distance computations from objects far away from the query can be avoided.

Exact kNN algorithms find the true k nearest neighbours. In some applications, it can be sufficient to find only an *approximate* result which has some small quantifiable error or an approximate result which is correct with some probability. An advantage of this approach is that an approximate algorithm can be faster, and it usually employs some indexing technique.

Another distinction is in query type. A *single-query* solution processes one query at a time. However, a significant increase in throughput can be gained if we use a *multi-query* approach. Since there are no dependencies between different queries, they can be processed in parallel. Furthermore, queries are often evaluated on the same database, so database objects, once loaded from memory, can be used for more than one distance computation. In some problems, the set of queries is the whole database. This is sometimes called All-kNN in literature, and it can be used, for example, in kNN graph construction.

The magnitude of k plays an essential role in kNN. Algorithms for small k may assume that the top k list fits into the cache or even into registers. For $k = 1$, the problem reduces to finding a minimum. In these cases, the overhead of updating an intermediate top k result could be pretty low, and implementation might be limited by other factors. On the other hand, for large k, the top k list does not fit into an on-chip memory (such as registers or local cache). It might be beneficial to use more sophisticated algorithms and data structures which would have prohibitively high overhead in the small kNN case. For large k, sorting the whole list of distances might be more efficient than using a complicated incremental approach.

1.2 Distance function

The efficiency of any kNN implementation is heavily influenced by the computational complexity of the distance function. Suppose a computationally cheap distance function is used. In that case, the computation might be limited by other factors, such as the speed of memory transfers or the selection of the top k distances. For expensive distance functions, top k computation and memory transfers take a relatively short time, so it is more important to speed up the distance computation or outright avoid some distance computations using indexing.

1.2.1 Cheap distance functions

The two most prominently used functions, Euclidean distance (also known as the L_2 distance) and cosine similarity (Definition 1.2; $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ denotes the dot product), are among the computationally cheap distance functions. Euclidean distance is a particular case of the more general L_p distance (Definition 1.1) for $p = 2$. If we set $p = 1$, we get the Manhattan distance. Another example of a common, cheap distance function is the Hamming distance which compares strings. Hamming distance between two strings is the number of positions in the two strings in which the strings differ.

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{1.1}$$

$$\frac{\langle \boldsymbol{x}, \boldsymbol{y} \rangle}{\|\boldsymbol{x}\| \cdot \|\boldsymbol{x}\|} \tag{1.2}$$

Since the computation of these distance functions requires only a small number of operations, it is usually limited by memory throughput. The multi-query approach is advantageous in these cases because a database object, once loaded in an on-chip memory (such as cache or registers), can be used for multiple distance computations, which limits the total number of memory transfers.

Matrix multiplication

Distance computation of the commonly used distance functions in a vector space can be viewed as matrix multiplication [32, 37]. Let $\boldsymbol{Q} \in \mathbb{R}^{q \times d}$ be a matrix of query vectors, and $\boldsymbol{P} \in \mathbb{R}^{p \times d}$ be a matrix of database vectors where q denotes the number of queries, p is the number of database vectors, and d is the dimension of all the vectors. A standard matrix multiplication $\boldsymbol{D} = \boldsymbol{Q}\boldsymbol{P}^T$ computes the dot product between the query and database vectors. The dot product can be viewed as a distance function since it is a cosine similarity without length normalization (Definition 1.2). However, matrix multiplication is not limited to distance functions derived from the dot product. For example, a matrix multiplication algorithm can compute the squared Euclidean distance if we replace the multiplicative operator with $\otimes(x, y) = |x - y|^2$. Several other commonly used distance functions can be obtained using a similar approach. The benefit of using matrix multiplication is that there are several well-optimized matrix multiplication libraries [7, 2].

1.2.2 Compute-intensive distance functions

Compute-intensive distance functions have much higher computational complexity. They can benefit from parallelization of a single distance computation and caching since a computation of a single distance usually requires significantly more passes over the data, unlike, for example, Euclidean distance, which can be computed in a single pass. Compute-intensive functions include, for example, Levenshtein's distance [16] which computes the distance of two strings as the minimum number of operations (insertion, deletion, and substitution of characters) required to transform one string to the other. Another example is the

Earth mover's distance [41] (EMD) designed to compare histograms, and Signature quadratic form distance (SQFD) [17]. Both SQFD and EMD can be used to compare signatures extracted from multimedia content such as images.

Indexing

Due to the complexity of distance computations, it is undesirable to compute these functions for each database object. Properties of the distance function can be used to quickly prune database objects which are too far from the query [21]. A typical assumption is that the distance function is a metric (Definition 1).

Definition 1. *Let M be a set and d a function $d : M^2 \to \mathbb{R}$ such that it satisfies the following properties. Then d is called a metric.*

1. $\forall x, y \in M : d(x, y) = d(y, x)$ *(symmetry)*
2. $\forall x, y \in M : d(x, y) \geq 0 \land d(x, y) = 0 \Leftrightarrow x = y$ *(non-negativity)*
3. $\forall x, y, z \in M : d(x, y) + d(y, z) \geq d(x, z)$ *(triangle inequality)*

For example, pivot-based methods [20] select a subset of database objects as pivots. Distances between pivots and the rest of the database objects are precomputed. The triangle inequality can be used to derive a lower bound of the distance between a query and a database point. For a query q, database object x, and a pivot p, it can be derived from triangle inequality and symmetry that $d(q, x) \geq |d(x, p) - d(q, p)|$.

Using a distance lower bound is the main idea of some popular indexing techniques for kNN, such as the Linear Approximating and Eliminating Search Algorithm (LAESA) [34]. In LAESA, a lower bound of a distance between a query q and a database object x is computed as $L(q, x) = \max_{p \in P}\{|d(x, p) - d(q, p)|\}$, where P denotes the set of pivots. The search algorithm has to compute distances between the query and all pivots $d(q, p)$ only.

The base LAESA finds only the nearest neighbour. The algorithm iterates over the database objects in order of increasing lower bound L. It maintains the nearest neighbour candidate, which is initialized to the nearest pivot at the start of the search. In each iteration, the distance to a database object is computed, and if the database object is closer than the current candidate, the candidate is replaced. The search ends when the algorithm encounters a database object whose lower bound L is higher than the distance to the nearest neighbour candidate.

LAESA can be easily extended to kNN [35]. k-LAESA maintains an intermediate top k result, and the stopping criterion is modified to compare the distance lower bound L with the actual distance to the kth nearest neighbour found so far.

In order to make the search easily parallelizable, prefiltering methods based on LAESA do not sort the database [27]. A sequential scan of the database is used instead, which computes the lower bounds and filters out objects too far from the kth nearest neighbour found so far. The distance function is computed for candidates who pass the first step only, and the computed distances are used to update the intermediate top k result. One issue with this approach is that, in many cases, a single distance computation does not fully utilize all processing cores. We have to compute multiple distances in parallel in order to achieve maximal throughput. Ideally, the intermediate top k result would be updated after each distance computation, but since this is not possible in a parallel implementation, it may lead to unnecessary distance computations.

Space-splitting indexing structures split the space (usually, a vector space is assumed) using a geometrical object. A kd-tree [39] is an example of an indexing structure for d-dimensional data. It is a binary tree where each node splits the vector space into two parts according to a selected axis (i.e., it uses axis-aligned hyper-planes to split the vector space). The splitting axes are cycled so that on level i of the tree, $i \mod d$ axis is used. Other notable indexing structures in this category include variants of the R-tree [19] like R^+-tree [43] or R^*-tree [15], which partition the vector space using hyper-cubes, and M-tree [21], which partitions the data using hyper-spheres.

1.3 Outline

The rest of this work is organized as follows. In Chapter 2, we introduce the GPU programming model and architecture. Chapter 3 analyzes kNN implementation, parallelization of the problem and mapping of kNN to GPU architecture. In Chapter 4, we explore several optimizations of kNN kernels. Chapter 5 evaluates GPU implementations of kNN for various configurations of the problem. We also evaluate the effectiveness of our optimizations and compare them with the state-of-the-art implementations. Chapter 6 summarizes our findings.

2. GPU Programming

In this Chapter, we describe principles of GPU programming relevant to an efficient, parallel k nearest neighbours implementation. The first section introduces the programming model and hardware architecture of GPUs. The rest of the Chapter focuses on details of running a code on a GPU and available APIs that can be used for communication and synchronization of GPU threads.

2.1 Architecture

This section introduces the programming model and architecture of GPUs. We decided to use CUDA terminology as it is the most prominent platform for general-purpose computations on GPUs at the time of writing. However, alternative platforms such as OpenCL or Vulkan have very similar concepts, which are named differently.

Graphics processing units (GPUs) are devices connected to a host system usually using a PCIe bus [3]. Originally, GPUs were intended for graphics computations but can also be used for general-purpose computations. GPUs usually have a large number of cores, but they are specialized for numeric computations. Unlike CPU cores, which dedicate many transistors for optimizing control flow using complicated scheduling, branch predictors, and large caches, GPU cores are much simpler in this regard.

GPUs are designed for data parallelism. The same code processes multiple data elements in a massively parallel fashion. This is implemented in practice by executing one function called a *kernel* on different data by many GPU threads. Compared to CPU threads, creating and scheduling GPU threads is more lightweight. Moreover, many more threads are used than on a typical CPU.

2.1.1 Thread hierarchy

While CPU cores work largely independently, GPUs take a different approach. Threads are assigned to GPU processors (called Streaming Multiprocessors, *SM*s) in large groups of threads called *thread blocks*. Each SM has its own registers, a shared L1 cache, and processing cores. Once a thread block is assigned to an SM, it runs its kernel until all threads in the block finish their computation.

The size of a thread block is limited because all threads assigned to an SM share resources of the processor. For this reason, a kernel can be executed by more than one thread block, which can run on different SMs in parallel. A group of thread blocks that execute the same kernel is called a *grid*.

CPU and GPU threads significantly differ when it comes to scheduling and running a code. A CPU thread is scheduled on a processor core where it runs for a while and then is preempted so that other threads can run. Different CPU cores usually execute different instructions. On a GPU, each thread block is assigned to an SM, and it is subsequently divided into smaller pieces called *warps*[1] which are scheduled on processing cores. Code is executed using the Single Instruction

[1]Warp size is usually 32 threads on modern GPUs.

Multiple Threads (*SIMT*) model. In SIMT, cores execute the same instruction, but each core has its own set of registers so that they can work on different data. When any thread is stalled, the whole warp is switched for another warp which is ready to run.

Code branches are supported by masking. All threads execute all divergent branches in a warp. Threads which should not execute a branch are disabled for all instructions in that branch. This situation limits parallelism since individual branches are executed by a warp serially. It can sometimes be avoided by reorganizing data to create a balanced workload for all threads and to avoid unnecessary data-dependent conditional statements within a warp.

Preemption of threads is another difference between CPU and GPU. Preemption of CPU threads is usually a somewhat complicated operation. On GPUs, thread blocks allocate all resources they need before execution, including registers. It limits the number of threads that fit onto an SM but allows SMs to quickly switch out a warp if an instruction stalls (e.g., due to a memory transfer or a complicated operation).

2.2 Memory hierarchy

A code running on a GPU cannot directly access data in a host system memory. Instead, all threads have to use *global memory.* Global memory serves a similar purpose to the main memory of a CPU. It is physically located on the GPU, and its size is in order of gigabytes. Data has to be transferred to global memory, usually from host system memory, via a PCIe bus before computation, and it has to be transferred back after computation. Data transfers to and from global memory can overlap with other computations on GPU.

Global memory is accessed in transactions which operate on large blocks of memory. Transactions have to be aligned to their size (i.e., the first address of a transaction has to be divisible by the transaction size). When a warp executes a global memory read or a write, an appropriate number of transactions is executed to cover all accessed addresses. In an ideal situation, all threads in a warp access subsequent addresses in an adequately aligned block of memory. A single global memory transaction is used in this situation. On the other hand, if threads in a warp access seemingly random addresses, multiple transactions have to be created. In this case, a large portion of data accessed by the transactions could be transferred in vain.

2.2.1 Registers and local memory

SMs have a large number of registers (thousands of 32-bit registers). The register file is the fastest memory available on a GPU. They are allocated and managed by the compiler. Variables in a kernel are usually stored in registers, but a compiler can decide to spill registers to local memory due to high register pressure.

Local memory is a part of global memory reserved for a thread. Large variables such as big structures or arrays can be stored in local memory instead of registers. Local arrays, which are addressed dynamically (i.e., the index of an element is only available at runtime), have to be stored in local memory because registers are not dynamically addressable.

2.2.2 Shared memory

Shared memory is a special memory located in each SM. It is similar to an on-chip CPU cache, but unlike CPU caches, it is programmer-managed. It has lower latency and higher throughput than global memory, so it can be used to manually cache frequently used values to avoid costly global memory transactions.

When a kernel is called, it requests a size of shared memory allocated for each thread block. Threads in a thread block can use shared memory to communicate. The memory is divided into banks (memory modules). Banks can be accessed in parallel. Accesses to a single bank, however, have to be serialized. Shared memory is mapped to address space so that consecutive words [2] are mapped to different banks. The number of banks is usually high enough for all threads in a warp to access their own bank. A *bank conflict* occurs when an instruction causes more than one thread from a warp to access different words from the same memory bank. However, if two or more threads from a warp request the same word, the instruction is only performed once, and the result is broadcasted to all requesting threads.

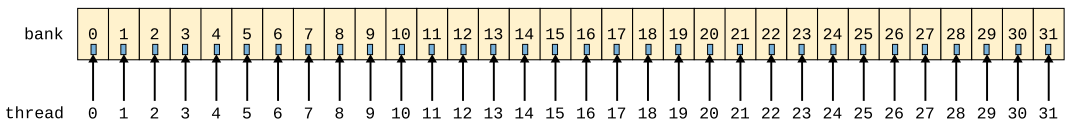

Figure 2.1: Linear access to shared memory banks (no bank conflicts).

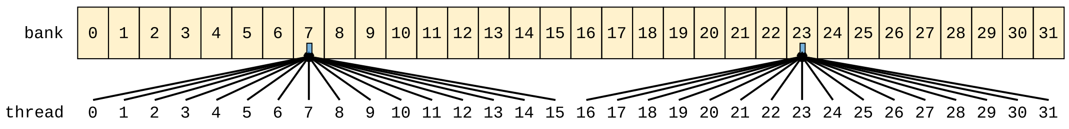

Figure 2.2: Broadcasts in shared memory (no bank conflicts).

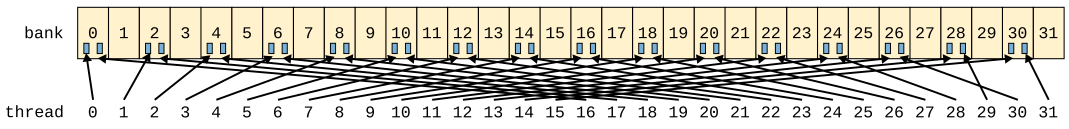

Figure 2.3: Strided access to shared memory.

Figure 2.3 shows an example of a bank conflict. Each thread accesses an element with index $2i$, where i is the thread index within its warp. Odd banks are not utilized in this example; even banks have to process two requests. Figure 2.1 shows an example of linear access in which all threads in a warp access a different bank. This example does not contain a bank conflict. A situation in Figure 2.2 also does not contain a bank conflict if all threads read the same address since the values from banks 7 and 23 are broadcasted to requesting threads.

2.3 Programming model

GPUs execute special C++ functions called *kernels*. Kernels are functions marked by an extended syntax of C++. When a GPU C++ compiler is used (e.g., the

[2]Usually, 32 or 64 bit words are used on modern GPUs.

CUDA compiler `nvcc`), it compiles these kernel functions separately to a code that can run on a GPU.

A kernel is executed in parallel by many threads. The total number of threads should far exceed the number of cores physically available on the GPU so that if some threads are stalled on an instruction, other threads can be scheduled instead. The number of threads that execute a kernel is usually determined by the input size (e.g., one thread could be used to process four input elements).

Listing 1 shows an example of a CUDA kernel which takes a list of floats as an input and computes the hyperbolic tangent of all elements in the input list. The `__global__` keyword marks the function as a kernel. Furthermore, each thread has a unique index accessible in code. The following predefined variables are available in a kernel:

- `threadIdx`: index of a thread within its thread block
- `blockIdx`: index of a thread block in the grid
- `blockDim`: number of threads in each thread block
- `gridDim`: number of thread blocks in the grid

In Listing 1, each thread is assigned one data element so its index can be computed using the formula in line 3.

```
1 __global__ void kernel(const float* input, float* output, int n)
2 {
3     int global_idx = threadIdx.x + blockIdx.x * blockDim.x;
4     if (global_idx < n)
5     {
6         output[global_idx] = tanh(input[global_idx]);
7     }
8 }
```

Listing 1: Example of a CUDA kernel that computes a hyperbolic tangent of a vector.

Threads are organized into up to three-dimensional thread blocks and grid for convenience (Figure 2.4). Hence we have to use the `x` property (the first dimension) of `threadIdx` and `blockIdx` in Listing 1.

2.3.1 Runtime API

Launching a kernel is very similar to calling a function. CUDA extends the standard C++ syntax of function calls to include configuration parameters. The configuration always contains the number of thread blocks and the number of threads in each thread block that will execute the kernel and a few other optional parameters. Listing 2 shows an example of launching the kernel from Listing 1.

Since GPUs cannot directly access system memory (Section 2.2), it is necessary to allocate global memory and transfer data from system (host) memory to the global memory of the GPU before running a kernel. CUDA API has

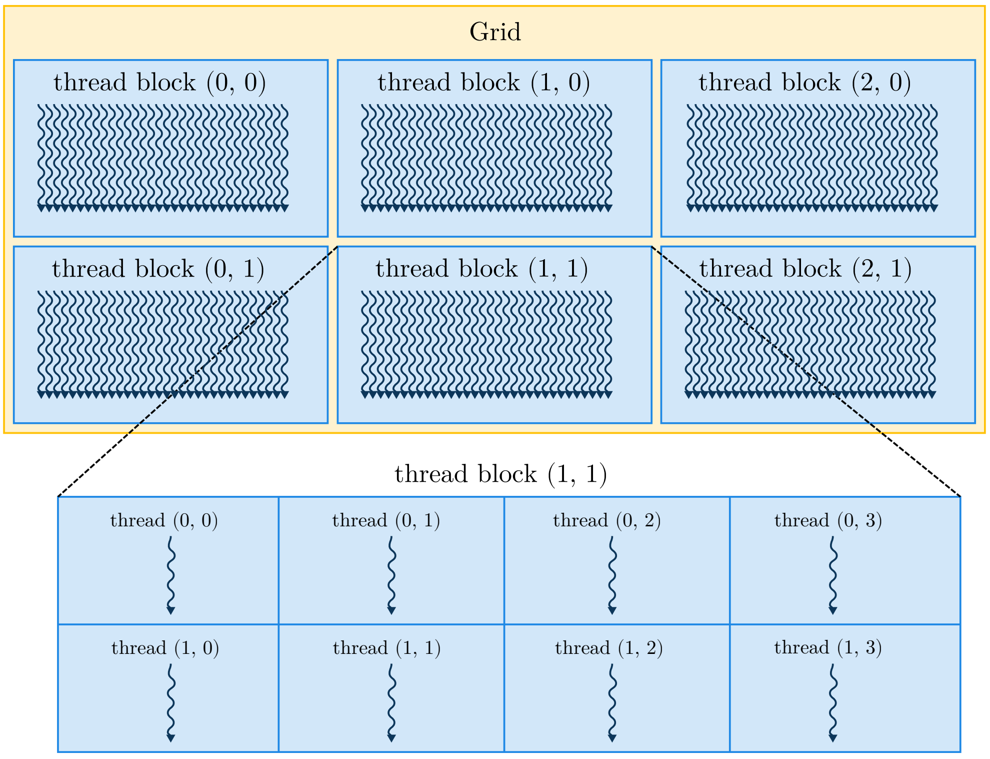

Figure 2.4: Thread blocks in a grid.

`cudaMalloc` and `cudaFree` functions to allocate and free a continuous block of global memory (lines 6 and 7 in Listing 2).

The `cudaMemcpy` function transfers data between the host and global memory. The last parameter of the function determines the direction of the transfer. The `cudaMemcpyHostToDevice` flag tells the function to transfer data from host memory to the GPU, and the `cudaMemcpyDeviceToHost` flag tells it to transfer data from the GPU to host memory. In Listing 2, the code at line 9 transfers the input to the GPU and the code at line 16 transfers the computed result to the host memory.

Most CUDA API functions return an error code. The last error code can be obtained by calling the `cudaGetLastError()` function. The `cudaGetErrorString` returns a string with a description of the given error. Listing 2 does not handle errors for brevity.

Kernels can also access *mapped memory*. Mapped memory resides in the host memory space but is also accessible from the GPU through a separate pointer. When a kernel accesses the memory, the data are transferred implicitly [3]. *Unified memory* is an alternative memory management technique. It provides a memory space accessible from GPUs and the host system. The data transfers are done automatically and on demand. We will not elaborate on this further as it is not relevant to our work. Additional details can be found in the CUDA Programming Guide [3].

```
1  float *gpu_input, *gpu_output;
2  int n = 1000;
3  std::vector<float> cpu_input(n);
4  std::vector<float> cpu_output(n);
5  // allocate global memory on the GPU
6  cudaMalloc(&gpu_input, n * sizeof(float));
7  cudaMalloc(&gpu_output, n * sizeof(float));
8  // transfer data from system memory to the GPU
9  cudaMemcpy(gpu_input, cpu_input.data(), n * sizeof(float),
10            cudaMemcpyHostToDevice);
11 // run 4 thread blocks, 256 threads in each one
12 int num_threads = 256;
13 int num_blocks = (n + num_threads - 1) / num_threads;
14 kernel<<<num_blocks, num_threads>>>(gpu_input, gpu_output, n);
15 // transfer the result from GPU to system memory
16 cudaMemcpy(cpu_output.data(), gpu_output, n * sizeof(float),
17            cudaMemcpyDeviceToHost);
18 // deallocate memory
19 cudaFree(gpu_input);
20 cudaFree(gpu_output);
```

Listing 2: Example of calling a CUDA kernel from a C++ code.

2.3.2 Host synchronization

The `cudaMemcpy` function is blocking. CPU thread is blocked until the memory is transferred. It has an asynchronous variant `cudaMemcpyAsync`. Asynchronous functions start the operation and then return before the operation finishes. Calling a kernel is an asynchronous operation. However, different CUDA calls are executed sequentially, so the memory transfer at line 16 in Listing 2 waits for the kernel to finish. The `cudaDeviceSynchronize` function can be used to wait for all preceding CUDA calls.

Modern GPUs are capable of executing several kernels at the same time. Data transfers to and from GPU and kernel execution can be performed concurrently. CUDA uses *streams* for overlapping work. A stream is a sequence of commands (API function calls like data transfers and kernel calls). Commands in the same stream are executed sequentially. However, commands from different streams can overlap. The `cudaStreamSynchronize` function waits for all preceding commands in the stream to finish but does not affect commands in other streams.

All CUDA functions which support streams use stream 0 by default. The default stream is unique in that adding a command to the default stream causes an implicit global synchronization. Preceding commands in all streams have to be finished, and subsequent commands in any stream have to wait.

Events provide another way to synchronize CUDA commands. Events are markers that can be used for timing and fine-grained synchronization within a stream or in between two streams. The `cudaEventRecord` function marks a location in a stream. Programs can wait for all commands preceding an event by calling the `cudaEventSynchronize` function.

2.3.3 Synchronization of GPU threads

Threads in the same thread block can communicate through shared memory (Section 2.2) and synchronize access using the `__syncthreads()` function. The `__syncthreads()` function is a barrier [3]. Threads have to wait until all other threads from their thread block reach this point in code. The function also acts as a memory fence. It makes sure that all shared and global memory operations made by the threads calling the `__syncthreads()` function are visible to other threads.

Warp-level synchronization

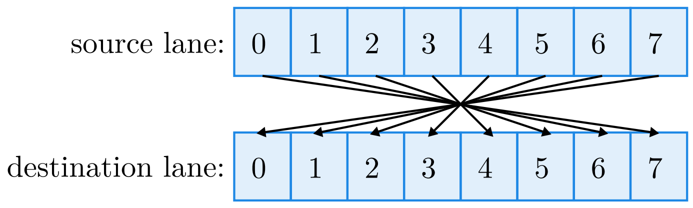

Figure 2.5: Example of a warp shuffle using `__shfl_xor_sync()` with mask 8.

Threads in a warp can communicate using warp instructions which should be faster than shared memory. *Warp shuffles* exchange variables between threads in a warp. To identify threads within a warp, they have a unique index called a *lane ID*. There are several warp shuffle instructions that differ in addressing. Figure 2.5 shows an example of a warp shuffle instruction that computes the target lane ID as a bitwise XOR with the current lane ID.

Another way to communicate within a warp is to use *vote functions*. Vote functions perform a reduction and broadcast to all threads in a warp. They all take a `mask` and a `predicate` as arguments. The `mask` is a bitmask with one for each thread that should participate in the operation. The `predicate` is an integer value that should be reduced. The following list summarizes available vote functions.

- `__all_sync` returns non-zero if and only if `predicate` is non-zero for all participating threads.
- `__any_sync` returns non-zero if and only if `predicate` is non-zero for any participating thread.
- `__ballot_sync` returns a mask with one for each participating thread for which the `predicate` is non-zero.

Atomic operations

Atomic instructions provide safe read-modify-write operations when multiple threads access the same address in shared or global memory. If two or more threads access the same address, an atomic operation will be finished before any other thread can access the address. CUDA provides basic operations such as addition `atomicAdd` (`atomicSub`), increment `atomicInc` (`atomicDec`), minimum `atomicMin` (`atomicMax`), and bitwise operations (`atomicAnd`, `atomicOr`, `atomicXor`). More complex operations can be implemented using a compare-and-swap instruction `atomicCAS(addr, compare, value)` which stores the `value` to the address `addr` if the old value at that address matches `compare`. All atomic instructions work on 32 or 64-bit integers. The addition also works on floats.

The performance of atomic instructions depends on the number of collisions. They perform well if threads mainly access independent addresses. However, their performance degrades if all threads access a shared address. On the other hand, a kernel that uses other communication mechanisms instead of atomics might also use more registers, resulting in worse performance.

Cooperative groups

Cooperative groups provide an abstraction for a group of threads. The abstraction includes functions for partitioning the groups, synchronization, and communication of threads in the group. Additionally, it provides a way to synchronize the whole grid from within a kernel. All functions and types are in the namespace `cooperative_groups`.

The `thread_group` type represents a generic group of threads. Its `sync` method synchronizes threads using an appropriate synchronization primitive (e.g., if the group represents a thread block, `__syncthreads()` will be used). The size of the thread group and index of the current thread within the group can be queried using the `size` and `thread_rank` methods, respectively.

There are types which represent specialized groups. The `this_thread_block()` function returns a group which represents the current thread block, `this_grid()` represents the whole grid, and `coalesced_threads()` represents currently active threads within a warp (i.e., non-disabled threads in a conditional branch).

Thread groups can be partitioned using the `tiled_partition(group, size)` function, which returns a group of threads within a warp. It has methods for warp shuffles and other warp communication mechanisms. The `mask` parameter of warp communication methods is derived from the thread group automatically.

2.4 Programming guidelines

In this section, we summarize the most important implications of GPU architecture for performance and provide general guidelines for GPU programming.

- Kernels should be executed with enough threads so that if a warp is stalled on an instruction, another warp can be scheduled instead.
- The size of shared memory and the number of registers limits the number of warps that can fit onto a single Streaming Multiprocessor (Section 2.1). Register and shared memory usage should therefore be limited. However, there is a tradeoff between the efficiency of memory access optimizations (which might use more registers or shared memory) and the latency-hiding mechanisms used by CUDA (which require more threads on the same SM), so careful benchmarking is necessary.
- Global memory is accessed in transactions (Section 2.2). Threads in warps should avoid random access to global memory. Ideally, a warp would access consecutive addresses in an adequately aligned memory block.

- Frequently used data can be manually cached in shared memory or registers, which are faster than global memory.
- Each thread in a warp should access its own bank when working with shared memory to avoid bank conflicts (Section 2.2). Consecutive words (usually, 32-bit words are used) are mapped to different banks. There are enough banks for all threads in a warp to access their own bank.
- We should try to organize data to avoid data-dependent branches in a warp and create a more balanced workload for each thread in a warp.
- Streams (Section 2.3.2) should be used to overlap data transfers with kernel execution.

We will refer to these guidelines when we describe optimizations of CUDA kernels in our work.

3. Analysis

In this chapter, we first define the problem of kNN. We introduce and analyze parallel kNN approaches. We focus especially on algorithms for GPUs.

3.1 Problem definition

The kNN problem can be formulated as follows. Given a collection of database objects $p_1, \ldots, p_n$ from a set D, a set of query objects $Q \subseteq D$, a distance function $d : D^2 \to \mathbb{R}$, and $k \in \mathbb{N}$, find k objects from $p_1, \ldots, p_n$ for each query $q \in Q$ with the smallest distance to q according to the distance function d. That is, for each query $q \in Q$, find different indexes $i_1, \ldots, i_k$ such that $d(p_{i_1}, q) \leq \cdots \leq d(p_{i_k}, q) \leq d(p_j, q)$ for all $j \in \mathbb{N}$, $1 \leq j \leq n$, $j \notin \{i_1, \ldots, i_k\}$.

3.1.1 Sequential solutions

Nearest neighbours can be found using a simple *sequential scan* algorithm. Each object is read from a database once, and a distance is computed from the object to a query. The distance, together with an object identifier, is inserted into a priority queue. At most k objects are maintained in the priority queue at a time. When it exceeds this threshold, the largest object (i.e., the object with the largest distance) is removed. After processing all database objects this way, the priority queue contains the k nearest neighbours of the query.

The priority queue is usually implemented as a binary or a d-ary heap, a generalization of a binary heap where each node has d children. An advantage of a d-ary heap is that the parameter d can be chosen so that all children of a node reside in the same cache line. A practical implementation might also use a simple sorted array as a priority queue if k is sufficiently small.

Another straightforward approach is to compute all distances and sort them. This solution is easy to implement since databases often have an efficient sorting procedure already. However, it can be very inefficient if k is much smaller than the number of database objects, which is often the case in many applications of kNN.

3.1.2 Indexing for nearest neighbours

An alternative way to answer kNN queries is to build indexing data structures. Section 1.2 mentioned space-splitting data structures, which can be used for spatial indexing. However, the performance of these data structures degrades for large dimensions due to the curse of dimensionality [46]. Even with moderately small dimensions $d > 10$, kNN queries in these structures visit most objects in the database. Moreover, due to a non-trivial overhead of search algorithms, queries often perform worse than a simple sequential scan.

Pivot-based methods have been used successfully for kNN indexing with compute-intensive distance functions [28]. These methods compute a lower bound of an actual distance between the database object x and a query object y. The lower bound is defined by $L(x, y) = \max_{p \in P}\{|d(x, p) - d(y, p)|\}$ where P is the set of all pivots. The $d(x, p)$ term can be precomputed. However, it is necessary to compute $d(y, p)$ for all pivots $p \in P$. For common distance functions such as Euclidean distance, lower bound computation and subsequent prefiltering have a cost comparable to directly computing the distance function $d(x, y)$ in many cases.

3.2 Parallelization

For the parallelization of kNN, we will look at the two parts of kNN (distance computation and k-selection) separately. Distances from a query have to be computed for all database objects if we do not use any precomputed indexing structure. Since there is a wide variety of distance functions, there is no general approach that would work for all of them. However, for the most common distance functions, such as Euclidean distance or cosine similarity, the problem reduces to matrix multiplication (Section 1.2) [32], so parallel matrix multiplication libraries can be used.

The goal of k-selection is to find k nearest neighbours given distances that have been computed already. For a multi-query problem, each query can be processed independently by a separate thread using the sequential scan approach. For a single-query problem, we could have one shared priority queue and use a similar algorithm. However, using one shared data structure would require a complex synchronization that would hinder the computation. The database is instead divided into equally sized parts. Each part is processed by a different thread independently of the other threads using a local priority queue. Once all threads finish their computation, the k nearest neighbours are found by merging the partial results.

3.3 Distance computation on GPUs

A GPU implementation of matrix multiplication can be modified to compute L_p distances or cosine similarities [32] instead of dot products. There are several matrix multiplication libraries for GPUs, such as CUBLAS [2] or MAGMA [7]. However, the CUBLAS library is not open source and its API is not flexible enough to allow for this change. The MAGMA library can be easily modified.

The modification replaces the $\times$ operator and possibly also the $+$ operator. The resulting matrix multiplication algorithm is unchanged otherwise. The entry in row i and column j of $A \cdot B$ where A and B are matrices of size $n \times k$ and $k \times m$, respectively, is $(A \cdot B)_{ij} = (A_{i1} \otimes B_{1j}) \oplus \cdots \oplus (A_{ik} \otimes B_{kj})$ where $\oplus$ is the new addition operator and $\otimes$ is the new multiplication operator.

For example, to compute Euclidean distances (or L_p distances for any p), we replace the $\times(x, y)$ operator with $\otimes(x, y) = |x - y|^p$. The $+$ operator remains unchanged. An important difference between a regular matrix multiplication and the modified version is that $\otimes(x, y)$ is not associative and does not distribute over the addition operator, an assumption often made in theory when working with matrices. However, practical implementations of matrix multiplication, like the one in the MAGMA library, usually have a far more relaxed set of requirements.

3.3.1 Distances based on a dot product

Euclidean distance and cosine similarity can be defined in terms of a dot product (Equation 3.1 shows the decomposition of Euclidean distance) [26]. Moreover, the dot product is the most computationally demanding operation for both functions since it has to be computed for all pairs of database and query vectors. An alternative approach to distance computation is to run a matrix multiplication kernel (e.g., using CUBLAS), which produces the dot products, and then run a kernel for postprocessing. The CUBLAS API also allows multiplying the result by a constant so the whole term $-2\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ can be computed using only a single CUBLAS call.

$$\|\boldsymbol{x} - \boldsymbol{y}\|^2 = \|\boldsymbol{x}\|^2 + \|\boldsymbol{y}\|^2 - 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle \tag{3.1}$$

Euclidean distance postprocessing includes L_2 vector norm computation ($\|\boldsymbol{x}\|^2$ and $\|\boldsymbol{y}\|^2$ from Equation 3.1) and the addition of the norms to the result of matrix multiplication. Since norm computation has to be done for only $n + q$ vectors where n is the number of database vectors and q is the number of query vectors, we can split the postprocessing into two kernels. One kernel computes the norms, and another kernel adds them to the result of matrix multiplication. Moreover, if only the ranking of objects is necessary as an output of kNN and not actual distances, we can avoid computing norms of query vectors since they will not change the order of kNN results [26]. Similarly, we can avoid computing the square root as it is an increasing function, so it does not change the order. The postprocessing for cosine similarity is almost identical, except the dot product is divided by the vector norms.

3.3.2 Specialized kernels

Other approaches in literature use specialized kernels for distance computation [31, 30]. Kuang et al. [31] partition the distance matrix into tiles which are assigned to thread blocks. It is similar to matrix multiplication kernels but does not cache subtiles in registers, unlike optimized matrix multiplication kernels such as the kernel from the MAGMA library [7].

A distance computation kernel due to Kruliš et al. [30] uses a different partitioning of the distance matrix. Each thread block computes distances from assigned queries to all database vectors so the thread block can immediately process queries. In its original formulation, it finds a minimum distance for k-means clustering, but it is possible to modify the kernel to compute k nearest neighbours instead. Furthermore, since the distances are processed in place, the distance matrix does not have to be stored in global memory.

The algorithm uses shared memory and registers for caching. Figure 3.1 summarizes the caching strategy. Each thread block starts by loading query vectors assigned to it from global memory to shared memory. Query vectors remain cached in shared memory for the whole duration of the computation (the shaded region in matrix W at the top of Figure 3.1). Database vectors are loaded to shared memory using a sliding window (the X matrix at the bottom of Figure 3.1).

The kernel is intended for L_2 distance computation. Similar distance functions (e.g., cosine similarity or L_p) can be computed with slight modifications. Each

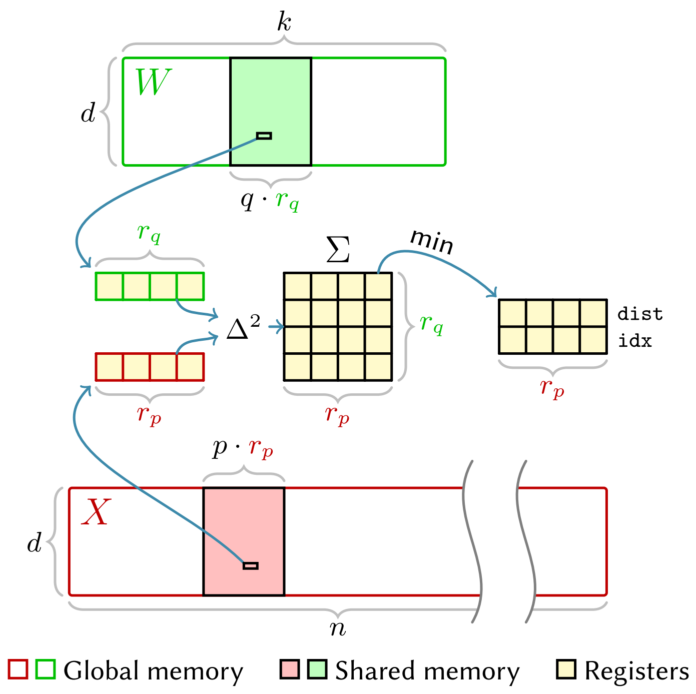

Figure 3.1: Caching strategy of a distance kernel implemented by Kruliš et al. [30].

thread computes an $r_q \times r_p$ submatrix of distances in registers. The r_p and r_q parameters are configurable template constants so that the distance submatrix of each thread can be stored in registers. The distance computation iterates over the vector dimension. For each dimension, r_q vector components from query vectors and r_p vector components from database vectors are loaded from shared memory to registers. For each pair of values, a square of a difference is computed and added to the corresponding entry in the distance submatrix (visualized in the middle of Figure 3.1). When all database vectors in shared memory are processed this way, the window moves to the next batch of database vectors.

3.3.3 Other distance functions

For computationally demanding distance functions, the computation time of distances dominates the overall computation time of kNN. Indexing and approximate algorithms are usually employed, which is beyond the scope of our work. However, for Euclidean distance and other typical distance functions, the k-selection step takes a significant portion of the overall computation time of kNN. We will therefore focus on the parallelization of this step in the following sections.

3.4 Selection using sorting on GPUs

Sorting can be used directly for k-selection (Section 3.2). However, even if it is not used to sort the whole list of distances, many CUDA implementations of kNN [26, 23, 29, 32, 44] use sorting as a fundamental building block. Hence, an efficient, parallel implementation is necessary.

A typical approach to parallel sorting on a CPU is to use a variant of the Merge sort algorithm. The input array is divided into small blocks sorted in parallel using an efficient, sequential sorting algorithm. Neighbouring blocks are then recursively merged until the whole input is sorted. An efficient GPU implementation has to use a parallel merge implementation that properly utilizes all GPU threads [42]. Furthermore, we can utilize fast on-chip memory to sort small arrays in the base case of Merge sort. Sorting networks like Bitonic sort have proven to be very efficient in this context [26, 31].

3.4.1 Bitonic sort

Sorting networks [13, 33] provide an alternative way to parallelize sorting. A fundamental building block of a sorting network is a comparator which implements a compare-and-swap operation. Given two inputs, the values are swapped if the first input is greater than the second input. Otherwise, the values are copied in their original order to the output. We visualize sorting networks using horizontal lines to represent values and vertical lines connected to two values to represent comparators (Figure 3.2).

Bitonic sort is a sorting network. Its fundamental idea is to sort Bitonic sequences (Definition 2), which can then be used to sort an arbitrary sequence. Bitonic sequences are sorted using a component called a *separator*. Separator gets as an input a Bitonic sequence $x_0, \ldots, x_{2n-1}$ and produces a permutation of this sequence $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$ such that:

1. $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$ are Bitonic sequences
2. $\forall i, j \in \{0, \ldots, n-1\} : a_i \leq b_j$

Definition 2. *Sequence $x_0, \ldots, x_{n-1}$ is Bitonic if there exists a rotation of the sequence such that a prefix of the rotated sequence is non-decreasing and the rest of the sequences is non-increasing. That is, there exists a $j \in \{0, 1 \ldots, n-1\}$ such that $x_j, x_{(j+1) \bmod n}, \ldots, x_{(j+n-1) \bmod n}$ can be split into two parts:*

- *a non-decreasing part of size k — $x_j, x_{(j+1) \bmod n}, \ldots, x_{(j+k-1) \bmod n}$*
- *a non-increasing remainder — $x_{(j+k) \bmod n}, \ldots, x_{(j+n-1) \bmod n}$*

In Figure 3.2, S_4 illustrates two separators for four values. A Bitonic sequence is sorted by a recursive splitting into smaller Bitonic sequences using separators until sequences of length one are reached. At this point, it follows from the properties of separators that the whole sequence is sorted. Moreover, all compare and swap operations within a separator are independent and can be done in parallel. Bitonic sequences can therefore be sorted using $\log_2 n$ layers of separators where n is the length of the input sequence (assumed to be a power of two).

A general sequence (i.e., not necessarily a Bitonic sequence) is sorted using an observation that two sorted sequences can be easily merged into one Bitonic sequence. If one of the sorted sequences is reversed and concatenated to the other, the result is a Bitonic sequence, and the process described in the previous paragraph can be used to sort such sequences. At the start, sequences of length one are trivially sorted. Neighbouring sequences are merged and sorted using the described algorithm, which produces sorted sequences of length two, and the whole process is repeated until the whole input is sorted. In total, the process has to be repeated $\log_2 n$ times where $n = 2^m$ is the length of the input, and each iteration has a logarithmic number of layers. The total number of separator layers can be computed using Equation 3.2.

$$\log_2 2^1 + \log_2 2^2 + \cdots + \log_2 2^m = 1 + \cdots + m = \frac{(1+m)m}{2} \in \Theta(\log_2^2 n) \quad (3.2)$$

We will denote by M_{2i} a merge component which, given two sorted sequences of size i, produces one sorted sequence of size $2i$. As discussed in the previous paragraph, M_{2i} is built using separator layers. However, one of the inputs to M_{2i} has to be reversed to make the input a Bitonic sequence. Instead of reversing one of the inputs, we can change the first separator layer of M_{2i} to use different indexing.

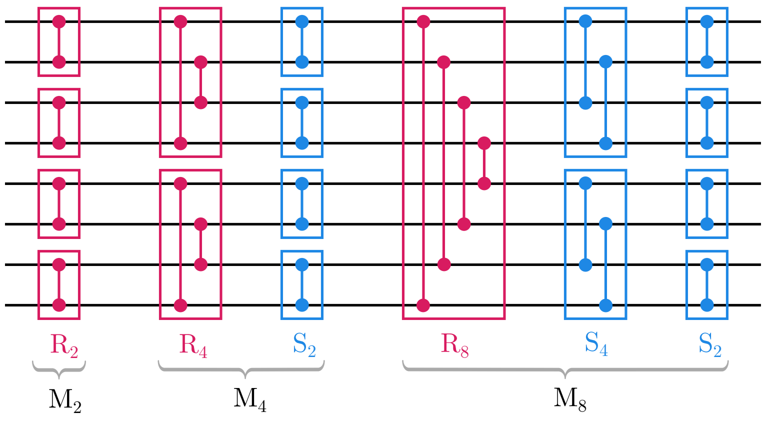

Figure 3.2: Bitonic sort for 8 values. The input is on the left.

Figure 3.2 shows the whole process of sorting eight values using Bitonic sort. The input is on the left of the figure. M_{2i} denotes a Bitonic merge component, S_j is a separator layer, and R_j is a reversed separator layer.

One disadvantage of Bitonic sort in the context of GPUs is that if the input array is too large to fit into registers or shared memory of a thread block, a large number of passes over the slow global memory have to be made (an order of $\log_2^2 n$) compared to other sorting algorithms. However, it has been used successfully to sort small arrays in one thread block [26, 44, 29] where a low latency, high bandwidth shared memory can be used or when the whole array fits into registers of one thread block.

3.4.2 Merge path

Merge path [38, 24, 32] is a parallel merge implementation. Assume we have two arrays sorted in ascending order, A and B. A sequential merge algorithm keeps a pointer to the first unprocessed element in A and B, respectively (initially set to the first element of each array). Elements at the pointers are compared, the smaller element is added to the output, and the corresponding pointer is incremented.

The Merge path algorithm looks at the sequential algorithm as a matrix (Figure 3.3). Each row corresponds to an element from sequence A, and each column

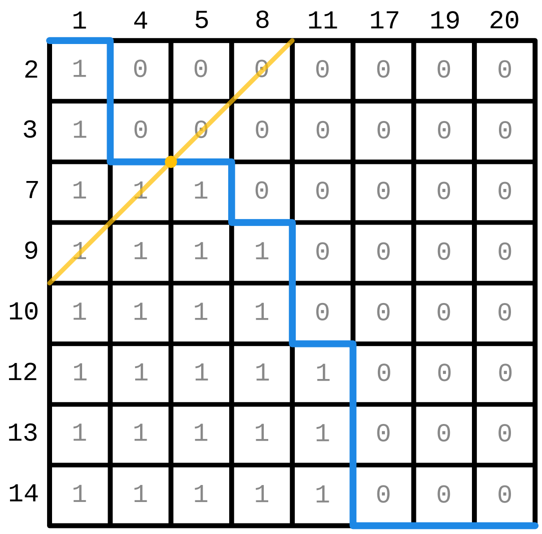

Figure 3.3: Intersection of a cross diagonal with a merge path in merge matrix.

corresponds to an element from sequence B. Progress of the merge algorithm can be viewed as a path starting in the top left corner of the matrix. If the current element from sequence A is greater than the current element from sequence B, the path moves to the right. Otherwise, the path moves down (the blue line in Figure 3.3). The path moves from the top left corner to the bottom right corner of the matrix.

If the matrix is filled with values of function 3.3, it is called a *merge matrix*. It can be seen that every *cross diagonal* (i.e., a diagonal which moves from the bottom left to the top right of the matrix; the yellow line in Figure 3.3) intersects the merge path at exactly one point.

$$M(i,j) = \begin{cases} 1 & A[i] > B[j] \\ 0 & otherwise \end{cases} \tag{3.3}$$

The Merge path algorithm does not explicitly compute the merge matrix. It is only helpful as a conceptual model. The parallel merge has two stages:

1. *Partition.* The merge matrix is partitioned using equally spaced cross diagonals. Each thread is assigned a cross diagonal and runs a binary search on this diagonal to find the intersection with the merge path.

2. *Merge.* Each thread uses the cross diagonal intersection from the previous stage as a starting point for a sequential merge.

The first stage essentially splits the merge path into equally sized, non-overlapping segments, so no synchronization is necessary for the second stage, and the workload is balanced among the threads.

Merge path implementation

The MGPU library [8, 14] implements the parallel merging of large arrays using the Merge path algorithm by running a coarse-grained partitioning stage in global memory. This stage splits the merge path into large, equally sized segments.

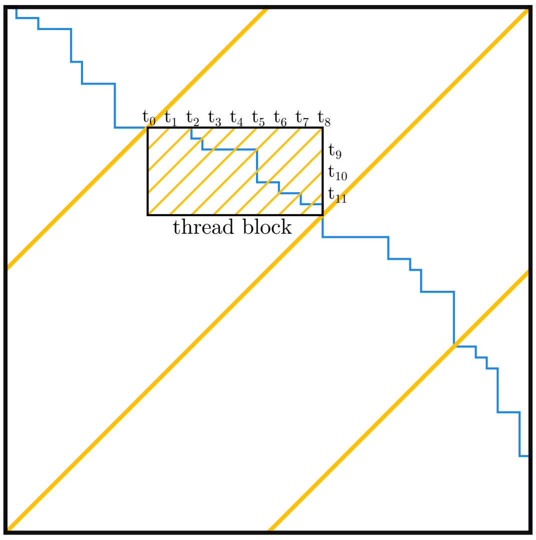

Figure 3.4: Coarse and fine grained partitioning of the Merge path.

Each segment is subsequently assigned to a thread block, and relevant parts of the arrays from global memory are loaded to shared memory. The Merge path algorithm is then used on the data in shared memory with much smaller spacing between cross diagonals. Each thread in this fine-grained partitioning stage finds the merge path intersection and runs the second stage of the Merge path algorithm (sequential merge) for a small number of steps.

Figure 3.4 illustrates this approach for a hypothetical thread block with 12 threads. The blue path is the merge path which is unknown at the start of the algorithm. The large yellow cross diagonals split the merge path to segments assigned to thread blocks. Figure 3.4 illustrates only one thread block, but each segment is assigned to a different thread block. Since the path intersection is known for each thread block's top left and bottom right corner, and the merge path only moves right or down, fine-grained partitioning can be done on much smaller cross diagonals in shared memory.

Parallel sorting based on the Merge path algorithm can be implemented using the merge sort approach. Small segments of the input array are assigned to thread blocks and sorted in registers or shared memory, for example, using Bitonic sort. Sorted segments are then repeatedly merged until the whole array is sorted.

3.4.3 Radix sort

Radix sort is an alternative to Merge sort. Unlike Merge sort, Radix sort does only a constant number of passes over the data assuming the size of keys is a constant. The general idea of Radix sort is to represent keys as d-digit numbers in radix r [42]. The input is sorted by each digit, starting from the least significant digit and ending with the most significant digit. The reason to represent keys in radix r is that each digit can have a small range. Values with a small range can be efficiently sorted using a Bucket sort:

1. *Count.* Compute the histogram of digit values (i.e., for each digit value, count how many keys have the same digit).

2. *Scan.* Compute an exclusive prefix sum of all histogram bins. The result of this operation is the offset of each bin in the output array.

3. *Scatter.* Write each value to the output array using the computed offset of its digit as a starting point. The relative order of values in each bin has to be preserved so that the sorting is stable.

To parallelize the first step of the Bucket sort algorithm using GPUs, we could have one histogram in global memory, assign non-overlapping segments of input to thread blocks, and let each thread block increment histogram buckets using atomic instructions. However, this approach would introduce a large number of collisions as the number of buckets is small. Instead, Satish et al. [42] propose to use privatized histograms. Each thread block computes its own histogram based on its assigned segment of the input in shared memory.

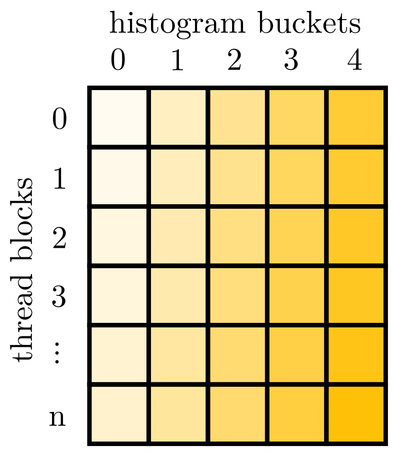

Figure 3.5: Histogram with 5 bins for each thread block in Radix sort.

The privatized histograms are written to global memory in a striped arrangement where all values of bin 0 are written to memory first, then values of bin 1, etc. (Figure 3.5; the histogram table is written to global memory in column-major order). An exclusive prefix sum of the whole histogram table is computed, which gives the algorithm a global offset of each thread block within each histogram bin. Prefix sum (scan) is a fundamental operation which can be parallelized. For example, the CUB library [1] provides several efficient, parallel CUDA implementations of a prefix sum.

The scatter step of Bucket sort, as we described it, could introduce a large number of global memory transactions per warp because neighbouring keys could easily fall into different histogram bins. Satish et al. [42] propose to split all values into small blocks. Each block is assigned to a thread block, and all values from the block are first scattered to shared memory. This operation groups values from the same bin to consecutive addresses in shared memory, which are written to global memory at the end of the scatter step. This reduces the number of discontinuities in a warp as consecutive threads are more likely to write values to the same bin and, thus, to consecutive addresses in global memory.

Parameter selection in radix sort

The choice of radix r affects global memory accesses. There is an inverse relationship between radix r and the number of Bucket sort passes over the slow global memory. On the other hand, as r increases, so does the size of privatized histograms, so more computation is needed for the prefix sum. Additionally, the coalescing issue discussed in the previous paragraph is more pronounced.

An optimal choice of radix r might not divide the key size evenly. In this case, a naive Radix sort implementation would effectively use a smaller r for the last iteration. However, using a drastically different parameter value for the last iteration might not be optimal. For example, the Radix sort implementation in the CUB library [1] uses two values for r (r and $r - 1$).

Radix sort for floating point numbers

For kNN, keys in Radix sort are often distances stored in a `float` or a `double` (using the binary IEEE 754 representation: sign bit, exponent, mantissa). Comparing positive floats using their binary representation as unsigned integers preserves the natural order of floats [25, 45]. If there are negative floats, we can use a one-to-one mapping m from 32-bit floats to 32-bit unsigned integers such that for all floats a, b, $a \leq b$ (comparison of floats) if and only if $m(a) \leq m(b)$ (comparison of unsigned integers).

The required mapping m always flips the sign bit; if the float is negative, it also flips all the other bits [25, 45]. The rationale for this mapping is that image of a positive number should always be greater than an image of a negative number. Since the sign bit is the most significant bit, we can achieve this by flipping the sign bit. One remaining issue is that more negative values have higher absolute, so the order of negative values has to be reversed. This can be achieved by flipping all bits of exponent and mantissa for negative floats.

3.5 Partial sorting

Partial sorting kNN algorithms utilize parallelized sorting to sort segments of size k. Neighbouring segments are merged, and only the lower half of the values from the two segments is kept in the next level. Each such iteration halves the number of values. After a logarithmic number of iterations, the algorithm produces the global top k values. Figure 3.6 illustrates this process for $k = 4$.

Li et al. [32] suggested implementing partial sorting using the Merge path algorithm (Section 3.4), which can be used directly to implement the merge steps. Other sorting algorithms, such as Bitonic sort, can be used to implement partial sorting.

3.5.1 Partial Bitonic sort

The partial Bitonic sort algorithm [29] utilizes Bitonic sort [13] for sorting and merging. Two sorted segments of size k can be merged using Bitonic sort by a logarithmic number of layers (Section 3.4.1, using a component we labelled M_{2k}).

The memory-optimized version of this algorithm [29] evaluates multiple kNN queries in parallel. It uses one thread block for each query. Each thread block

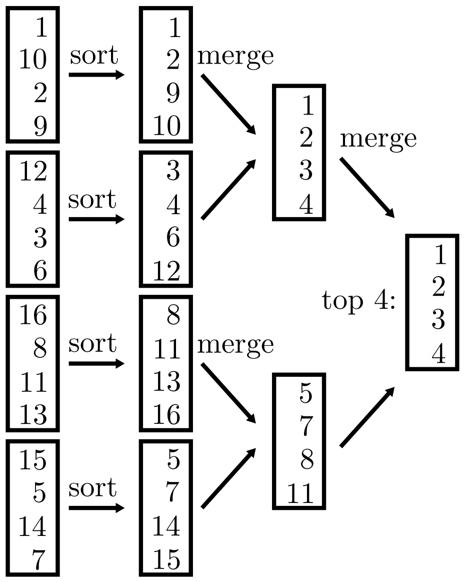

Figure 3.6: Parallel partial sorting method which finds the top 4 values.

keeps two segments of objects of size k in shared memory. The first two segments are initially loaded from global memory to shared memory and sorted using Bitonic sort. The rest of the input is processed in the following way:

1. All objects in the two sorted segments in shared memory are split into the lower and upper halves using the reversed separator R_{2k} from Bitonic sort (Figure 3.2).
2. The second segment, which now contains k objects with the highest distance, is replaced with a new segment from global memory.
3. Both segments are sorted using Bitonic sort. The first segment is a Bitonic sequence because it is the result of R_{2k} (this follows from the properties of separators in Section 3.4.1). The first segment can therefore be sorted by $\log_2 k$ layers of separators. The order of the second segment can be arbitrary, so a full Bitonic sort has to be used.

These steps are repeated until the whole input is processed, at which point, the top k distances are in the first segment. This version of the algorithm exposes less parallelism than what would the approach depicted in Figure 3.6 allow. However, the whole computation can be done in fast shared memory instead of global memory.

3.5.2 Limitations of partial sorting methods

Partial sorting exposes a large degree of parallelism. However, k-selection is a memory-bound problem. Reducing the number of slow global memory accesses at the cost of fewer opportunities for parallelism may be worth it. Furthermore, partial sorting may potentially do a lot of unnecessary work. Threads do not use an intermediate top k result to avoid unnecessary computation. Consequently, input segments with comparatively large distances have to be sorted and merged (e.g., the bottom two segments in Figure 3.6).

3.6 Incremental selection

Incremental k-selection maintains an intermediate top k result. This section focuses on the multi-query k-selection (Section 1.1). The main reason for this is that incremental solutions often require fast shared memory to store the intermediate result and lightweight synchronization. A thread block is the largest group of GPU threads with these characteristics. Because one thread block would not saturate all cores of a GPU, the multi-query approach is required to achieve high throughput.

Single-query problems can be adapted for the multi-query algorithms discussed here. The general strategy is to split the list of objects into blocks. Each block is assigned to a thread or a group of threads (depending on an algorithm, a group of threads could be anything from a warp to a whole thread block). Each thread or a group of threads updates its intermediate top k result. The partial results are merged when all threads finish their computation.

3.6.1 Data parallel selection on GPUs

The most straightforward adaptation of sequential k-selection is to assign each query to a GPU thread. Each thread keeps its own intermediate top k result in a priority queue. The priority queue contains the k nearest neighbours when the computation is done.

Distance matrix memory layout

If one thread is used per query, consecutive threads load distances from different queries. The distance matrix (Section 1.2) has to be stored in a column-major layout for optimal global memory utilization. For database objects $p_1, \ldots, p_n$, all distances from p_1 to all queries are stored in memory first, then distances from p_2, etc. If the columns are aligned properly (Section 2.4), all values from global memory transactions are utilized with this layout. The alternative row-major layout is the worst possible memory layout for this problem, as each thread would have to create its own global memory transaction with every read instruction.

Priority queue memory layout

The priority queue can be stored in shared memory or registers if k is small. However, for large k, there is not enough memory in a thread block for all priority queues. Even if all priority queues fit into shared memory or registers, excessive memory usage limits the number of thread blocks per SM, which hurts performance (Section 2.4).

An alternative is to store the priority queues in global memory. An optimal memory layout depends on the data structure used to implement the priority queues. If we use a simple sorted array, a column-major layout, like in the previous section, is optimal because reading several priority queues by a warp is coalesced.

Issues with a GPU implementation

One of the main disadvantages of the naive data parallel approach on GPUs is that it suffers from high thread divergence (Section 2.4). It can easily happen that only one thread from a warp has to insert a new object into its priority queue. At the same time, the remaining threads have to wait as the distance they loaded from global memory is larger than the largest distance in their priority queue so far. More sophisticated incremental kernels hence try to utilize a larger group of threads (at least a warp) to answer a query with one shared priority queue for the whole group.

3.6.2 Merge queue

The merge queue algorithm [44] addresses issues of the data parallel approach (Section 3.6.1) by using the whole thread block for each query. It maintains a queue of top k objects, which is shared by all threads in a thread block.

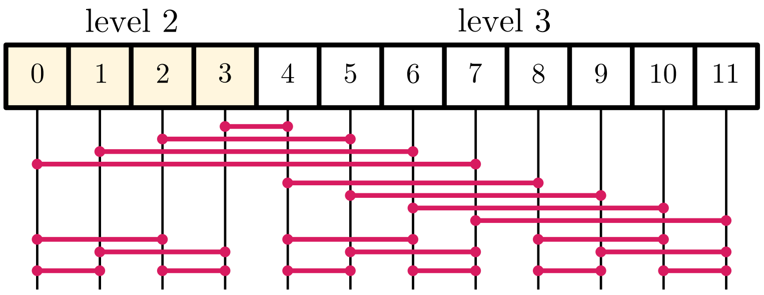

Figure 3.7: Merging two Merge queue levels of size 4 and 8 using Bitonic sort.

Merge queue is a priority queue implementation designed for parallelization on GPUs. It is a hierarchical data structure with multiple levels. Each level is a sorted list (in decreasing order). The first level and the second level have the same size, which we denote by m. m is a parameter of the algorithm, and we assume it is a power of two. Subsequent levels have size $m \cdot 2^{l-2}$ where $l > 2$ is the level (i.e., the size of each subsequent level is double the size of the previous level).

Merge queue has two invariants:

1. Distances at each level are sorted in decreasing order.
2. Level heads (the largest object at each level) are sorted in decreasing order.

Objects are inserted into the first level using the insertion sort approach. The largest object on the level is pushed out. Note that the largest object on the first level is also the largest object in the whole data structure (this follows from the Merge queue invariants). If an insert breaks the second invariant, all objects on both levels are merged using Bitonic merge (Section 3.4.1). Figure 3.7 illustrates this operation for two levels of different sizes. No additional memory is necessary for merging because Bitonic sort can be implemented as an in-place sorting algorithm. However, merging two levels can break the invariant on lower levels, so it has to be checked and fixed recursively.

Figure 3.8 shows an example. We try to insert 36 into the queue.

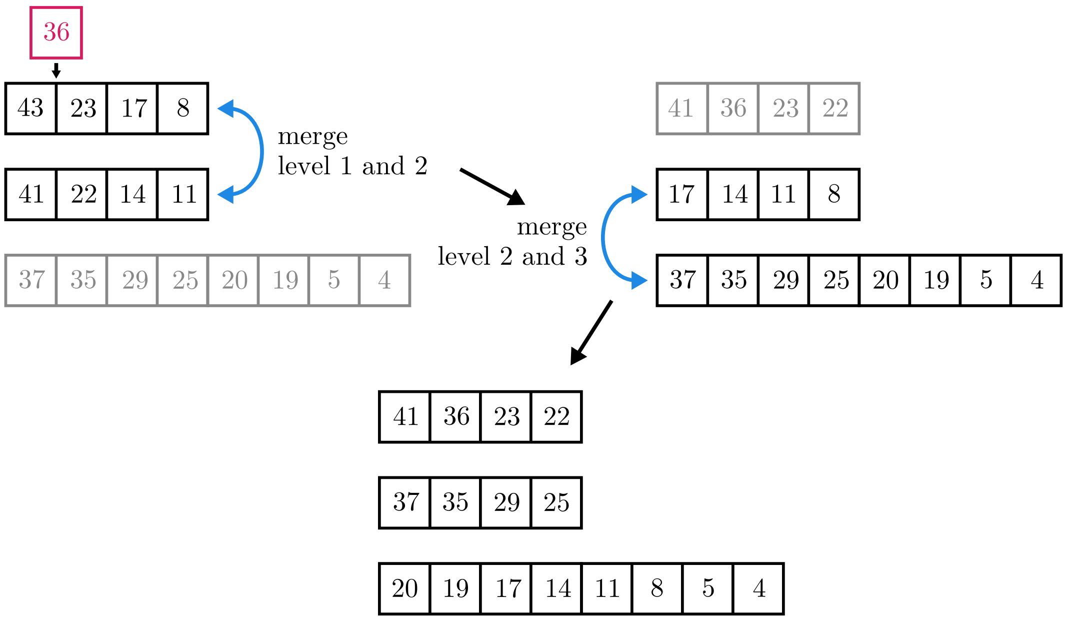

Figure 3.8: Fixing Merge queue invariants after insertion.

1. The algorithm inserts 36 to the second position and the largest object on the first level (43) is removed.
2. 36 is now the largest object on the first level, so the algorithm has to check the second invariant. Because $36 < 41$, the Merge queue invariant is broken. The first two levels are merged.
3. After this merge, the largest object on the second level (17) is smaller than the largest object on the third level (37), so the algorithm has to merge levels two and three.
4. After this operation, the data structure invariant is fixed and the algorithm ends.

Buffered search

Tang et al. [44] first insert all objects into a fixed-sized buffer. When the buffer fills up, it is sorted, and objects from the buffer are inserted one by one into the Merge queue, starting with the smallest object. If an object in the sorted buffer is larger than the largest object in the Merge queue, the insert operation can stop without processing the rest of the buffer. The kernel can thus avoid inserting objects that would be almost immediately removed if it did not use a buffer.

3.6.3 Warp Select

The Warp Select method [26] uses a warp for each query. The intermediate state is a warp-wide, sorted register array called a *warp queue*. Each thread from a warp stores several values from the warp queue. Additionally, threads maintain a *thread queue* which is also stored in registers. The thread queue is a sorted list of candidates. It serves as a buffer for new values.

Thread queues are sorted from the largest value to the smallest value. The warp queue is sorted in the opposite direction (the smallest values first). Moreover, the whole data structure maintains the following invariants [26].

1. The largest object in each thread queue is not in the top k.
2. The largest object in each thread queue is greater than all objects in the warp queue.
3. All processed objects so far, that are in the top k, are either in the warp queue or in a thread queue.

The list of objects is processed according to the following algorithm to maintain these invariants. Figure 3.9 shows a visualization of this process.

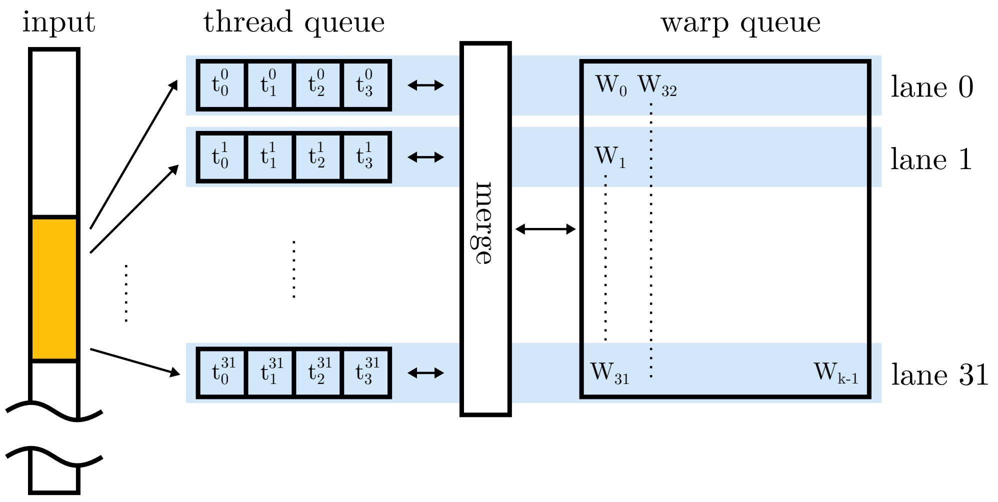
input
thread queue
warp queue
t_0^0 t_1^0 t_2^0 t_3^0
t_0^1 t_1^1 t_2^1 t_3^1
t_0^{31} t_1^{31} t_2^{31} t_3^{31}
merge
W_0 W_{32}
W_1
W_{31}
W_{k-1}
lane 0
lane 1
lane 31

Figure 3.9: WarpSelect overview [26].

1. Threads in a warp load consecutive objects from global memory.
2. Each thread compares its loaded object v with the first object in its thread queue t (i.e., the largest object in its thread queue).
3. If v is smaller than t, it is inserted into a proper position in a thread queue so it is still sorted. This operation removes the largest value, t, from the thread queue.
4. Each thread checks the second invariant. Threads use the ballot instruction to communicate whether any of them broke the invariant.
5. If any thread from the same warp breaks the invariant, the thread queues are interpreted as one big warp-wide register array, and it is sorted using Bitonic sort (Section 3.4.1). The sorted objects from all thread queues are then merged with the warp queue using Bitonic merge. After this operation, the warp queue has the k smallest objects, and thread queues contain the remaining objects. Finally, thread queues are reversed so that they are sorted in descending order.

The warp queue $W_0, W_1, \ldots, W_{k-1}$ is split among threads so that consecutive threads store consecutive values. The thread in lane i is assigned values $W_{i+32 \cdot j}$, where j is an index of the value within the lane, and 32 is the warp size (warp queue in Figure 3.9). Compare-and-swap operations in Bitonic sort are performed using warp shuffle instructions. For Bitonic sort strides that are a multiple of the warp size, the values are available locally in each thread, so no synchronization is necessary.

Warp Select uses a modification of Bitonic sort, which can sort arrays of sizes that are not a power of two. The modification is derived by padding the array so that its size is a power of two and using a standard Bitonic sort algorithm, as described in Section 3.4.1. Any compare-and-swap operation which has as an input a value from the padding can be left out. This modification is helpful in Warp Select because it allows for an arbitrary thread queue size which is vital for performance.

3.7 Selection

Selection techniques are mostly probabilistic algorithms to find the kth smallest distance in an unsorted list in linear time on average. Probably the most well-known algorithm in this category is the Quickselect algorithm. Once the kth smallest distance is known, the top k objects (distances with corresponding identifiers) can be retrieved trivially by scanning the list of objects. In this section, we will discuss adaptations of the Quickselect approach for parallel architectures and for GPUs in particular.

3.7.1 Sample Select

The Sample Select [40] method is an adaptation of the Quickselect algorithm for GPUs. The general idea is very similar to Bucket sort (Section 3.4.3), summarized in the following list.

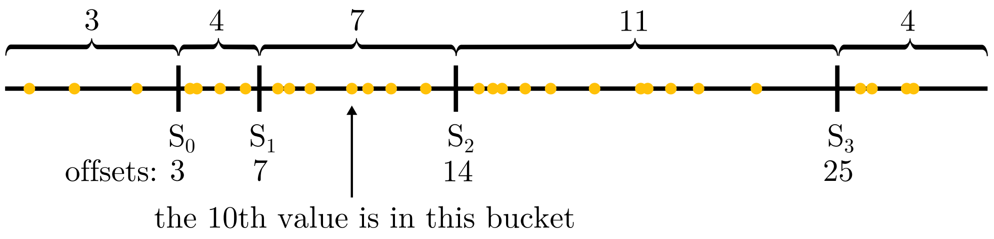

Figure 3.10: Sample select iteration with 4 splitters and $k = 10$.

1. *Pick splitters* from the list of objects and sort them.
2. *Count.* The splitters partition $\mathbb{R}$ into buckets (Figure 3.10). Assign each object to a bucket and count how many objects fall into each bucket.
3. *Scan.* Compute the prefix sum of histogram buckets.
4. *Select.* Find the bucket with the kth object (using the prefix sum), subtract the bucket offset from k, and repeat these steps with objects from the bucket.

The splitter search in the *Count* step can be parallelized using a data-parallel approach. Each thread is assigned an object, and it uses a binary search on the sorted array of splitters (in shared memory) to determine the bucket of the object. Each thread block has its own privatized histogram to avoid global synchronization. Threads increment histogram bins in shared memory using atomic instructions.

The privatized histograms are written to global memory in the *Scan* step in column-major order, and a parallel prefix sum implementation is used to find the offset of each bucket, just like in Radix sort (Section 3.4.3). Unlike the Radix sort, we are only interested in one bucket in the *Select* step (the bucket with the kth smallest object). Objects from this bucket are written to consecutive addresses in global memory using offsets computed in the previous step, and the whole process is repeated recursively.

Small lists in the base case are sorted using a parallel implementation of Bitonic sort (Section 3.4.1) within one thread block. In the context of GPUs, we consider a list to be small if it fits into shared memory or registers of a thread block.

Partitioning

In an ideal situation, the splitters in the *Pick splitters* step would produce approximately equal-sized buckets. The optimal ith splitter, s_i, is i/b percentile, where b is the number of buckets. The percentiles are approximated by selecting a random sample from the list of distances. The sample is sorted, and appropriate percentiles from the sample are chosen as splitters. Random samples can be selected on GPUs using the cuRAND library [4]. An issue with this approach is that the memory access pattern is inherently random, so, in general, a new transaction has to be created for each global memory read. However, we assume that the number of splitters is small, so the overhead of inefficient global memory accesses is minimal.

The splitters have to be searched to identify the bucket for each object in the *Count* step of the algorithm. A binary search can be used to find the bucket for each value because the splitters are sorted. However, since the binary search index calculations are complicated, Ribizel et al. [40] proposed to build an implicit, complete binary search tree in shared memory. The nodes are organized in an array as in a binary heap. The node at index i has its children at indices $2i + 1$ and $2i + 2$. Leaf nodes in this tree represent different buckets.

3.7.2 Radix Select

Radix sort (Section 3.4.3) represents objects (distances in case of kNN) as d-digit numbers in radix r. Radix digits partition objects to buckets similarly to splitters from the Sample Select algorithm [12]. Radix Select works the same as Sample Select (Section 3.7.1), except it uses radix digits to partition keys to buckets instead of splitters. The most significant radix digit is used in the first iteration. At the end of each iteration (the *Select* step from Sample Select), the kernel finds the bucket (radix digit) with the kth smallest object and recursively searches objects in this bucket. Since the most significant radix digit is the same for all objects in this bucket, the next radix digit is considered for partitioning in the recursive call.

Small lists (lists which fit into registers or shared memory of a thread block) in the base case are sorted using Bitonic sort (Section 3.4.1). The distance of the kth smallest object can be found trivially in this case. If the list of objects is too large to fit into a thread block, even after considering all radix digits, the algorithm stops. In this case, there are several objects with the same distance from a query, and we can choose any of them as the kth smallest.

An advantage of Radix partitioning is that a bucket for each object can be identified in constant time. However, bucket distribution depends on the distribution of distances. If the whole dataset consists of objects with a small distance, for example, the first few iterations will needlessly iterate over the whole dataset, as the most significant digit would be the same in this example for all objects. This can be partly prevented if the kernel keeps track of the largest distance in the list and rescales all distances if necessary. However, bucket size distribution may still be imbalanced if the dataset is skewed towards particular radix digits.

3.8 Conclusion

We analyzed several approaches to the parallelization of both parts of kNN (distance computation and k-selection) on GPUs. Computation of the most common distance functions can be reduced to matrix multiplication using either reexpression of the distance function in terms of a dot product (Section 3.3.1) or modification of matrix multiplication kernels (Section 3.3). We can use highly optimized matrix multiplication kernels for both approaches. However, specialized kernels developed specifically to compute Euclidean distances (Section 3.3.2) can be used to answer kNN queries with only one fused kernel.

A naive implementation of k-selection using sorting is inefficient if k is much smaller than the database size, as is often the case in many kNN applications in practice. Partial sorting methods (Section 3.5) try to address this inefficiency by only sorting blocks of size k. Incremental selection algorithms (Section 3.6) are a further optimization of partial sorting which avoids some expensive merge operations by maintaining an intermediate top k result. As an alternative method, algorithms derived from the Quickselect algorithm (Section 3.7) can be used to find the kth smallest object quickly. A simple filtering pass over the database can then be used to retrieve all top k objects.

4. Optimizations

This chapter discusses some optimizations of kNN we explored. In particular, we propose a kernel for small k-selection, a kernel for small k-selection fused with distance computation, and a kernel for large k.

4.1 Single-pass selection

In this section, we describe our multi-query, small k-selection kernel (assuming $k \leq 2048$). We focus on multi-query instances as they allow us to achieve high throughput using a fast incremental algorithm. However, this approach can also be used to parallelize single-query problems (Section 3.6). Moreover, because we assume a small k, we can find k nearest neighbours in a single pass (i.e., each distance is read from global memory exactly once). We start by introducing the general idea of our kernel. Further sections discuss implementation details.

The kernel is derived from the partial sorting method proposed by Kruliš et al. [29]. We take inspiration from the Merge queue algorithm [44] (Section 3.6.2) and the Warp Select algorithm [26] (Section 3.6.3) for optimizations. Each query is processed by a single thread block. Threads load distances from the distance matrix. If the loaded distance is smaller than the kth smallest value found so far, it is inserted into a shared buffer (which resides in shared memory) using the `atomicAdd` instruction. When the buffer fills up, it is merged with the intermediate top k result. The kth smallest distance used for prefiltering is updated whenever the buffer is merged. After reading all distances from the input, the kernel runs one final merge to get the final result.

We tried several parallel strategies for inserting objects (distance, label pairs) into the shared buffer. One approach is to count objects that have to be inserted in each thread and run a parallel (thread block-wide) prefix sum to find an offset in the shared buffer. However, this solution performs worse than a simple atomic add on a shared variable because the number of objects inserted into the buffer is typically small.

Compared with the other multi-query, small k-selection kernels, using one big, shared buffer has the advantage of aggregating objects for the merge operation. Instead of doing several inserts like in the Merge queue, all objects in the buffer are inserted simultaneously using a parallelized Bitonic merge. Moreover, the buffer is only merged with the intermediate result if it is filled with new objects. In Warp Select, thread queues have to be merged whenever an invariant of any thread queue is violated. An advantage of thread queues is that they are local, so operations with thread queues require no synchronization. Moreover, the intermediate top k result is potentially updated more frequently, so the distance used for prefiltering can be more effective. On the other hand, the size of thread queues is limited, it allocates many registers, and frequent merging can waste a lot of work if most thread queues still contain a lot of old objects when the merge occurs.

4.1.1 Bitonic sort optimizations

Sorting and merging the buffer with an intermediate top k result is computationally the most intensive operation of this kernel. A rudimentary implementation of a thread block-wide Bitonic sort can keep all values in shared memory. Each compare-and-swap operation on the data first loads both distances to registers compares them, and writes the result back. If the values are close together (within one warp), Bitonic sort steps can be implemented using warp shuffle instructions.

One issue with this implementation is that all operations on the data have to load values from shared memory to registers, execute the operation, and store the result back. This is inefficient because one stage of Bitonic sort might produce a result that the next stage could directly use without using shared memory.

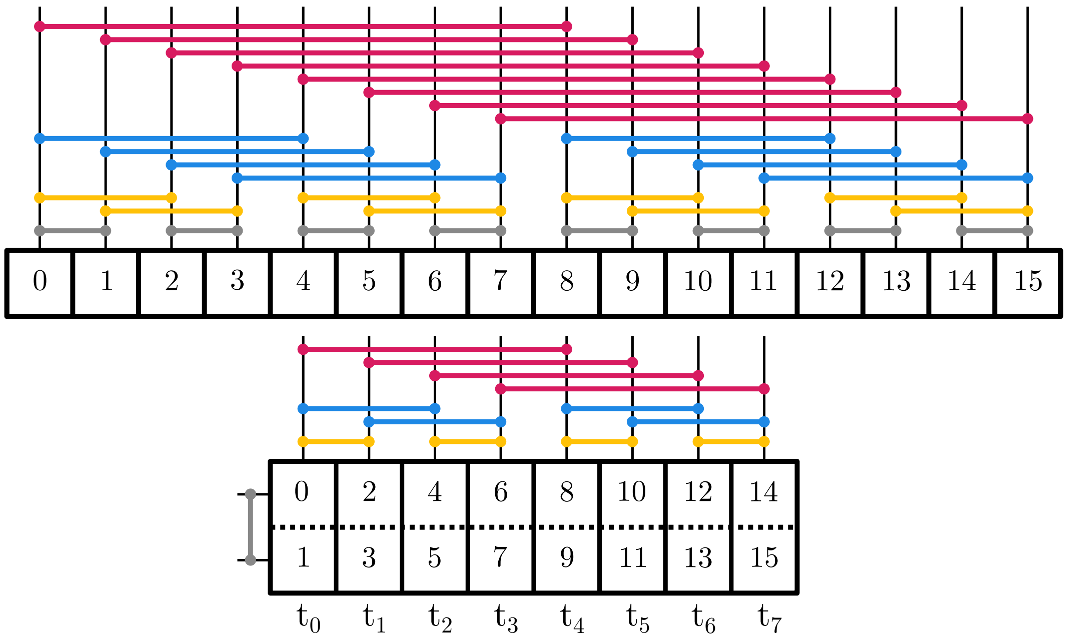

Figure 4.1: Mapping of Bitonic sort steps (S_{16}, S_8, S_4, S_2) to thread arrays with 8 threads.

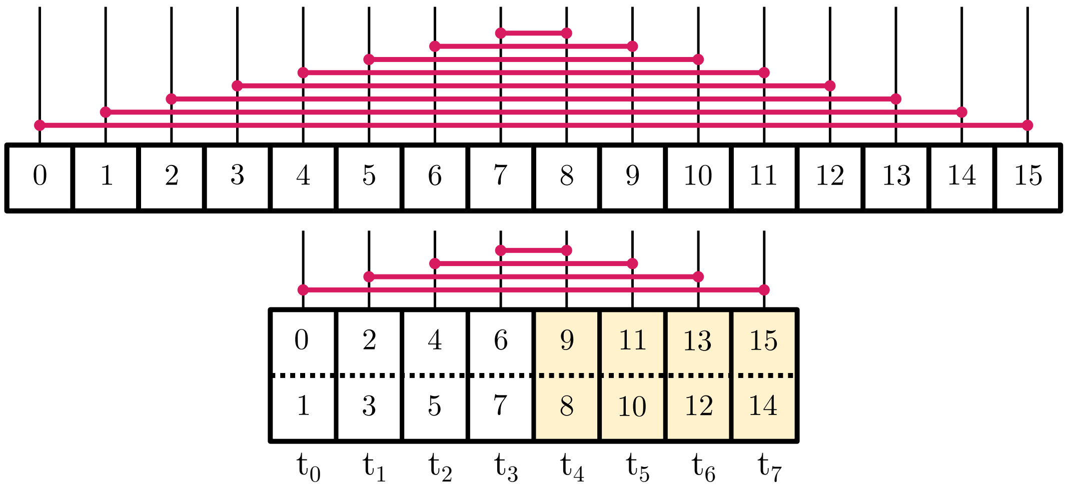

Figure 4.2: Mapping of the first reversed Bitonic sort stage (R_{16}) to 8 threads.

In our kernel, the whole array is stored in registers of a thread block similar to Warp Select. Unlike Warp Select, each thread stores several *consecutive* values from the array. Figure 4.1 shows the mapping of Bitonic sort steps to thread arrays. Bitonic steps on the lowest levels are performed on thread local data by each thread without synchronization. Higher levels use warp shuffles or shared memory if the Bitonic sort stride exceeds the boundaries of warps.

The reversed separator from Bitonic sort can similarly be mapped to thread arrays (Figure 4.2). However, some thread arrays have to be reversed so that the values align correctly for the operation (all arrays in the yellow-shaded region are reversed). Specifically, an array has to be reversed if it participates in a compare-and-swap operation as the second operand. This operation cannot be solved simply by different indexing in code because thread arrays must be addressed using compile-time constants. Hence, we reverse some thread-local arrays before the operation to avoid branching within a warp during computation.

4.1.2 Shared memory mapping

Some large Bitonic sort strides cannot be implemented with warp shuffles because we use the whole thread block to sort the arrays (buffer and the top k result). We could store the whole array in shared memory for large strides and run the Bitonic steps there. However, since each thread typically stores more than one value from the array, it could be processed in parts to reduce the shared memory usage of the kernel.

Figure 4.1 shows our mapping of objects from an array (top of the figure) to thread local arrays (the table at the bottom of the figure). Each column of this table is a thread-local array of objects. The rows of this table contain objects from different threads. If each thread stores t objects, the shared memory usage of the kernel can be reduced by a factor of t by using the following observation. For strides larger than or equal to t, Bitonic operations can be used on the ith row (the table at the bottom of Figure 4.1) independently of Bitonic operations on the other rows.

For example, consider eight threads, each of which stores two objects (Figure 4.1). The array and compare-and-swap operations of the Bitonic sort are depicted at the top of the figure. The table at the bottom of the figure shows the same operations. Only one row of this table has to be in shared memory at once. However, a thread compares and swaps two objects from each row. In order to employ all threads from a thread block, we store two consecutive rows from the table to shared memory. Each thread block is thus required to allocate space for $2b$ objects in shared memory, where b is the number of threads in a thread block. In our small example, we would have to load the whole table to shared memory. However, in general, the table could be bigger, and the kernel would still only have to allocate space for $2b$ objects.

Our implementation does not have any shared memory bank conflicts. We write the grid rows to shared memory one after the other, so thread i writes a value to index i and $i + b$. Bitonic sort steps in shared memory also do not introduce any bank conflicts because the smallest Bitonic sort stride we use in shared memory is the warp size, and thus threads from the same warp load consecutive objects.

4.1.3 Transposed memory layout

Each thread stores consecutive elements in the layout we described. Alternatively, we could transpose the whole memory layout. In the transposed layout, thread i stores elements $a_i, a_{i+b}, a_{i+2b}, \dots$ where b is the thread block size. A similar layout is notably used in the Warp Select kernel (Section 3.6.3).

An advantage of the transposed layout is that writing the registers to global memory in sorted order is coalesced. It works well in the Warp Select kernel because all threads are in the same warp. One disadvantage of the transposed layout with the whole thread block is that it forces the kernel to use shared memory for small strides.

Let w be the warp size, b the thread block size, n the length of the input list to be sorted (assumed to be a power of two), and $t = \frac{n}{b}$ the number of items per thread (size of thread local arrays). Bitonic separators S_i or R_i (Section 3.4.1) where $w < i \leq b$ have to be processed in shared memory in the transposed layout. For comparison, in the layout we propose, separators S_i or R_i such that $w \cdot t < i$ have to be processed in shared memory. Separator S_i or the reversed variant R_i is used in Bitonic sort $\log_2 \frac{n}{i} + 1$ times (Figure 3.2). Hence it is better to use shared memory for large strides as they are less common in Bitonic sort. Our proposed layout is thus better for our kernel in this regard. A disadvantage of the proposed layout is that writing values to global memory is not coalesced. However, we only write the array to global memory once at the end of the kernel, and this inefficiency was insignificant in our tests.

4.1.4 Bitonic sort implementation details

Algorithm 1 shows our implementation of Bitonic sort. It implements the steps depicted in Figure 4.1 (i.e., it sorts Bitonic sequences). All `for` loops have to be unrolled so that the thread arrays are addressed using compile-time constants and can be stored in registers. The t, w, s parameters are powers of two, and we provide them as template parameters so that they are compile time constants as well.

Parallelism is hidden in Bitonic sort steps in the innermost body of the `for` loops. Bitonic sort steps for small strides are implemented using warp shuffle instructions. Value of the thread with lane ID i is compared with the value of the thread with lane ID $i \oplus j$, where $\oplus$ denotes the bitwise XOR operation and j is a stride from Algorithm 1. The values can be exchanged between threads using, for example, the `__shfl_xor_sync` intrinsic. Both threads in lanes i and $i \oplus j$ perform the same comparison and replace their local value if necessary.

In shared memory, each thread loads and compares two distances. Contrary to the warp shuffle operation in the previous paragraph, the index of a thread within its thread block is treated as an index of a compare-and-swap operation rather than an index of an element. Let i be the thread index within its thread block, j a Bitonic sort stride from Algorithm 1 (hence, j is always a power of two), and a mask $m = j - 1$. Then each thread has to compare and swap values at indices $L = ((i \mathbin{\&} {\sim}m) \ll 1) \mid (i \mathbin{\&} m)$ and $R = L + j$ where $\sim$ denotes bitwise negation, $\mid$ denotes bitwise OR, and $\ll$ is a bit shift to higher order bits. The expression for L inserts a 0 bit at the $\log_2 j$th position in a binary representation of i. The rationale is that with stride j, we can group the array values to blocks of size j. If we number these blocks starting from 0, every value from an even block is used as a first operand of a compare-and-swap operation. This can be seen, for example, on the blue operations in Figure 4.1.

Algorithm 1: Optimized Bitonic sort steps $S_{2s}, S_s, S_{s/2}, \dots, S_2$

```
Input: items per thread $t$, warp size $w$, stride size $s$
Input: subsequences of size $s$ are Bitonic
Result: subsequences of size $2 \cdot s$ are sorted
if $s \geq t \cdot w$ then                                   // use shared memory
    for $i \leftarrow 0; i < t; i \leftarrow i + 2$ do
        store row $i$ and $i + 1$ to shared memory   // grid in Figure 4.1
        for $j \leftarrow s/t; j \geq w; j \leftarrow j/2$ do
            run Bitonic separator with stride $j$ ($S_{2j}$) in shared memory
        load row $i$ and $i + 1$ from shared memory back to registers
$s' \leftarrow \min\{\frac{s}{t}, \frac{w}{2}\}$
if $s' \geq 1$ then                                          // use warp shuffles
    for $i = 0; i < t; i \leftarrow i + 1$ do
        for $j \leftarrow s'; j \geq 1; j \leftarrow j/2$ do
            run Bitonic separator on row $i$ with stride $j$ ($S_{2j}$) using warp
              shuffles
$s' \leftarrow \min\{s, \frac{t}{2}\}$
for $j \leftarrow s'; j \geq 1; j \leftarrow j/2$ do
    run Bitonic separator with stride $j$ ($S_{2j}$) on the thread local array
```

Algorithm 2 implements the reversed Bitonic sort separator (Figure 4.2). The index of the first operand of each compare-and-swap operation is computed in precisely the same way as in Algorithm 1, and it is denoted by L. Index of the second operand is computed as $L \oplus (2s - 1)$ where s is a stride from Algorithm 2.

4.1.5 Global memory throughput

Our k-selection kernel is memory bound. It spends a significant portion of time waiting for global memory reads. In order to fully utilize global memory bandwidth, it has to keep enough memory transactions in flight.

Each thread issues multiple read instructions in each iteration to achieve high global memory utilization. Any instruction that would use the result is moved after this step, so a warp can only be stalled on a global memory read after it starts several transactions. This solution requires additional registers to hold the results of multiple reads.

We also tried to start memory transactions using the prefetch instruction (`prefetch.global.L2`) in addition to the previous solution. The instruction loads the requested address to the L2 cache, so no additional registers have to be allocated for the result before it is needed. L2 cache is shared by all SMs on the GPU, and its size is rather small. However, this approach could serve at least a small portion of global memory transactions from the L2 cache.

Algorithm 2: Reversed Bitonic sort separator R_{2s}

```
Input: items per thread $t$, warp size $w$, stride size $s$, ID of this thread
       within the thread block $tid$
Input: subsequences of size $s$ are sorted
Result: subsequences of size $2 \cdot s$ are bitonic
if $s \geq t \cdot w$ then                                   // use shared memory
    if $tid \,\&\, \frac{s}{t} \neq 0$ then
        reverse register array
    for $i \leftarrow 0; i < t; i \leftarrow i + 2$ do
        store row $i$ and $i+1$ to shared memory  // grid in figure 4.2
        run reversed Bitonic separator with stride $s/t$ ($R_{2s/t}$) on the rows
        load row $i$ and $i+1$ from shared memory back to registers
    if $tid \,\&\, \frac{s}{t} \neq 0$ then
        reverse register array
else if $s \geq t$ then                                      // use warp shuffles
    $lane \leftarrow tid \bmod w$
    if $lane \,\&\, \frac{s}{t} \neq 0$ then
        reverse register array
    for $i \leftarrow 0; i < t; i \leftarrow i + 1$ do
        run reversed Bitonic separator with stride $s/t$ ($R_{2s/t}$) on row $i$
          using warp shuffles
    if $lane \,\&\, \frac{s}{t} \neq 0$ then
        reverse register array
else                 // local operations without any synchronization
    run reversed Bitonic separator with stride $s$ ($R_{2s}$) on the local array
```

An alternative way to achieve high global memory utilization is to spawn more threads per SM. If a warp is stalled on a global memory read, SM can schedule another warp which starts the next memory transaction. However, we cannot afford to do that in our kernel because the number of threads per SM is limited by the high number of registers used by each thread.

4.1.6 Analysis

The majority of the work of our kernel is in adding values to the shared buffer and merging the buffer with an intermediate top k result. In this section, we compute an upper bound on the expected number of values added to the buffer during the whole computation assuming a random permutation of the input list of distances.

Complexity analysis is very similar to the analysis of the Warp Select [26] method, but it is easier because there is only one shared buffer. Let $d_1, \dots, d_n$ be a random permutation of the list of distances. In the following analysis, we find the top k distances from $d_1, \dots, d_n$ using the described kernel with a buffer of size k.

Observation 1. *If a distance d_i is added to the buffer, it is in the top $2k$ of $d_1, \dots, d_i$.*

Proof. If $i \leq 2k$, the implication is satisfied trivially. For $i > 2k$, assume, for a contradiction, that the rank of d_i is greater than $2k$ in $d_1, \dots, d_i$ and it is added to the buffer. Denote by d_m, $m < i$, the last element where a buffer merge occurred and r the kth smallest value in $d_1, \dots, d_m$. Note that such an m exists because, technically, the kernel merges the buffer after it reads the first k distances. There can be at most k values smaller than r in $d_{m+1}, \dots, d_i$ as we assumed that m is the latest merge point. Hence, the rank of r in the larger list $d_1, \dots, d_i$ can be at most $2k$. Since the kernel only adds values smaller than the current radius to the buffer, the rank of d_i in $d_1, \dots, d_i$ has to be less than or equal to $2k$, which is a contradiction. $\square$

Let I_i denote an indicator that d_i is in the top $2k$ of $d_1, \dots, d_i$ (i.e., $I_i = 1$ if d_i is in the top $2k$ of $d_1, \dots, d_i$, and it is zero otherwise). Probability $P(I_i = 1)$ is given by Equation 4.1. The first $2k$ values are in the top $2k$. For subsequent values, exactly $2k$ elements from i are in the top $2k$.

$$P(I_i = 1) = \begin{cases} 1 & i \leq 2k \\ 2k/i & i > 2k \end{cases} \tag{4.1}$$

Let N denote the total number of values added to the buffer during the whole computation. Using the previous observation, we can bound N from above as $N \leq \sum_{i=1}^{n} I_i$. Equation 4.2 computes expected value of the upper bound using linearity of expectation. H_n denotes the nth harmonic number defined as $H_n = \sum_{i=1}^{n} 1/i$.

$$\mathbb{E}N \leq \sum_{i=1}^{n} \mathbb{E}I_i = 2k + 2k \cdot \sum_{i=2k+1}^{n} 1/i = 2k \cdot (1 + H_n - H_{2k}) \tag{4.2}$$

For this analysis, we also want to compute M, the total number of buffer merges. Since the buffer is merged whenever it is filled, the expected number of merges, $\mathbb{E}M$, can be computed using Equation 4.3. We use $\ln n$ as a rough approximation for the nth harmonic number H_n in the last step.

$$\mathbb{E}M = \mathbb{E}\lceil N/k \rceil \leq 1 + 2 \cdot (1 + H_n - H_{2k}) \approx 3 + 2\ln(n/k) \tag{4.3}$$

The expected number of buffer merges is in $O(1 + \log(n/k))$, and the expected number of values added to the buffer is in $O(1 + k\log(n/k))$. If k is much smaller than n, adding values to the buffer is infrequent, which might be the reason why our strategy of allocating buffer slots using atomics on a shared variable performed better than other more complicated mechanisms.

4.2 Fused distance computation with selection

The k-means clustering kernel implemented by Kruliš et al. [30] (Section 3.3.2) performs a computation similar to k nearest neighbours in a single kernel call. It computes Euclidean distances and then finds a minimum for k-means clustering. Instead of finding a minimum, we can use the approach from the previous section for k-selection. That is, we add the computed distances to a shared buffer. When the buffer fills up, it is merged with the sorted top k array.

The original kernel computes distances from several queries in a single thread block. In order to preserve this flexibility in our modification, we place several buffers and top k arrays in one thread block. All buffers are sorted and merged with the top k arrays whenever any buffer fills up. This complicates the implementation of the kernel, and it allocates a lot of additional registers depending on the value of k. However, we observed that the kernel performs better with multiple queries per thread block.

Our Bitonic sort implementation can be used to sort several small arrays in parallel. The arrays are treated as one big, block-wide register array, and all Bitonic sort procedures stop at a certain stride length (when the stride reaches the size of the small arrays). An advantage of this implementation is that, for a small k, Bitonic sort steps are implemented using only thread-local operations and warp shuffles. This is an additional benefit of the memory layout we use for sorting in Section 4.1.3.

We store the buffers in a row-major layout in shared memory (i.e., all values from the first buffer are stored at the lowest addresses, followed by all values of the second buffer etc.). Depending on the parameters of the kernel, each thread has to load several consecutive values from a buffer during the merge operation. If a thread loads t values, it can cause a t-way bank conflict with this layout. However, indexing in an alternative memory layout that is bank conflict-free is significantly more complicated. The more straightforward layout with bank conflicts performs on par with the conflict-free layout or even better in some instances.

4.2.1 Applicability

An obvious advantage of the fused kernel is that the distance matrix is never stored in global memory. Depending on the dimension of input vectors, the size of the database, and the number of queries, the distance matrix can easily be the largest single piece of memory needed for kNN computation. Especially considering that dimension of input vectors should be fairly small since Euclidean distance is not a good distance function for high-dimensional data [11, 46]. As a result, the fused kernel has much lower memory footprint and can handle larger instances than the kernel from the previous section in some cases. Furthermore, distance computation and k-selection are done using a single kernel invocation.

A disadvantage is that the kernel uses a large amount of shared memory and registers. The shared memory requirements grow with input dimension and with k. The kernel is mainly suited for vectors with small dimensionality. Moreover, multiple queries have to be processed by each thread block. Because register requirements grow with k, we found that the fused kernel is mainly suited for small $k \leq 64$.

4.3 Multi-pass selection

Previous kernels cannot be used for large k when the top k list does not fit into a thread block. Other completely different solutions are required if k is given as a percentage of the total number of database objects. However, sorting the whole database is excessive even in these cases.

We adapted the Sample Select [40] method (Section 3.7.1) for large k, multi-query top k-selection. Compared to our single-pass kernel, multi-pass selection has to read some distances from global memory several times. It is, therefore, inherently slower than a single-pass selection since the problem is limited by memory throughput, but it works for much larger k.

Similarly to our small k-selection kernel, each query is assigned to one thread block. The kernel has two stages. The first stage finds the kth smallest distance, and the second stage gathers all objects in the top k (distances and object indices) in an output array. We use a single kernel call for both stages. In the first stage, distances are partitioned into buckets using Sample Select iterations:

1. Select a random sample of distances and pick splitters from this sample.
2. Create a histogram of buckets from the distances (splitters from the previous step define buckets).
3. Find the bucket which contains the kth smallest distance.
4. Subtract the bucket offset from k and repeat these steps recursively with the distances in the bucket from step 3.

We use the cuRAND library [4] to select a random sample (step 1) which is sorted using our implementation of Bitonic sort. The splitters are selected the same way as in Sample Select (i.e., we pick appropriate percentiles from the sample as splitters).

We stop the recursion when all distances fit into a thread block. All distances are sorted using Bitonic sort in the base case. The kth smallest distance can be found trivially in the sorted array. The second stage is a simple filter pass over the data. It finds all top k objects given the kth smallest distance found by the first stage.

The result of the described kernel is unsorted. Its size can be larger than k if there is more than one value with the same distance. Radix sort (Section 3.4.3) implementation (e.g., from the CUB library [1]) can be used to sort the top k objects. We use Radix sort in our implementation because it uses a constant number of passes over global memory, unlike, for example, Merge path or Bitonic sort.

4.3.1 Partitioning

The first stage is a recursive search for the kth smallest distance. It has to write all distances from the target bucket to consecutive addresses in global memory (step 4) so that the search can continue recursively. However, it cannot overwrite distances in the input array because they are used in the second stage. We use an auxiliary global memory buffer. The first iteration reads from the input array and writes the distances to the auxiliary global memory buffer. All subsequent iterations work in the buffer only.

In step 4, all distances are first added to a shared memory buffer. When the buffer fills up, it is written to global memory. The size of the shared memory buffer is larger than the size of a global memory transaction. All global memory writes are thus coalesced.

Buffering values

We use a thread block-wide prefix sum from the CUB library to allocate positions in the shared buffer. Each thread counts how many values it has to insert into the buffer. An exclusive prefix sum of these values gives the kernel an offset in the shared buffer for each thread.

Alternatively, buffer positions could be allocated using `atomicAdd` (Section 2.3.3) on a buffer size in shared memory. However, if the bucket with the kth value is large, most threads will have to insert a value into the buffer, and the performance could degrade due to a large number of collisions.

Temporary storage

An auxiliary global memory buffer of the size of the input is needed for the partitioning. The first stage cannot overwrite the input array with distances because the second stage uses the values. However, the auxiliary memory can be merged with the output array so that only $O(n-k)$ additional memory is needed per query where n is the database size.

4.3.2 Single-stage selection

It is possible to find k nearest neighbours in a single stage. The modified kernel differs from the two-stage approach in step 4. It partitions the whole input (distances and labels) into two parts. One part contains objects from the bucket with the kth smallest distance, just like in the previous kernel. The other part contains objects from all lower buckets.

A disadvantage of this approach is that the partitioning cannot be implemented with a simple buffered write, as in the previous section. Moreover, labels have to be moved together with distances. In the two-stage kernel, most of the computation (the first stage) works with distances only, so we can afford to allocate larger buffers and more registers to optimize memory accesses of the kernel better. We implemented both approaches, and we will evaluate them in the next chapter.

5. Evaluation

In this chapter, we evaluate current state-of-the-art exact kNN GPU kernels. The two parts of kNN (distance computation and k-selection) are evaluated separately except in the case of fused kernels, which compute both parts in a single kernel call. We also evaluate the effectiveness of optimizations proposed in the previous chapter.

5.1 Methodology

All kernels were evaluated on an NVIDIA Tesla V100 SXM2 32 GB (Volta architecture with compute capability 7.0). We used CUDA version 12.0 on a Rocky Linux server with a 16 core processor Intel Xeon Gold 5218. All tests record the execution time of CUDA kernels.

We repeat each measurement 20 times. The first ten measurements are discarded as a warmup. We use an arithmetic mean to combine the measurements. Results are mainly presented as a throughput (number of distances computed per second in case of distance kernels and number of distances processed per second in case of k-selection kernels). To evaluate our optimizations, we show the speed-up of the optimized kernel when compared with a baseline without the optimization. Speed-up is computed as the arithmetic mean of baseline measurements divided by the average of measurements of the optimized kernel. For both speed-up and throughput, higher values indicate a better solution. All error bars presented in this chapter show one standard deviation. Our measurements should only be used to compare GPU kernels within one figure.

The tested dataset is a (uniformly) random dataset on a d-dimensional cube $[0, 1]^d$. When we compare several kernels, we use the same random generator seed for all of them to test all kernels on the same input.

5.2 Common distance functions

In this section, we evaluate distance computation kernels discussed in Section 3.3.

- A specialized kernel proposed by Kuang et al. [31] (Section 3.3.2), which partitions the distance matrix to tiles assigned to thread blocks. Tiles from the input matrices are cached in shared memory.
- A distance computation method using the cuBLAS library (Section 3.3.1), which expresses the distance function in terms of a dot product. The dot product can be computed using a matrix multiplication kernel. This approach requires two additional kernels. One kernel computes the norms of all database vectors, and another kernel adds the norms to the distance matrix. Norms of query vectors are not computed (i.e., the kernel computes $\|\boldsymbol{x}\|^2 - 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ where $\boldsymbol{x}$ is a database vector and $\boldsymbol{y}$ is a query vector).
- A modified matrix multiplication kernel from the MAGMA library, which directly computes squared Euclidean distances. We use a kernel implemented by Li et al. [32]. However, we modified the kernel so that it can be

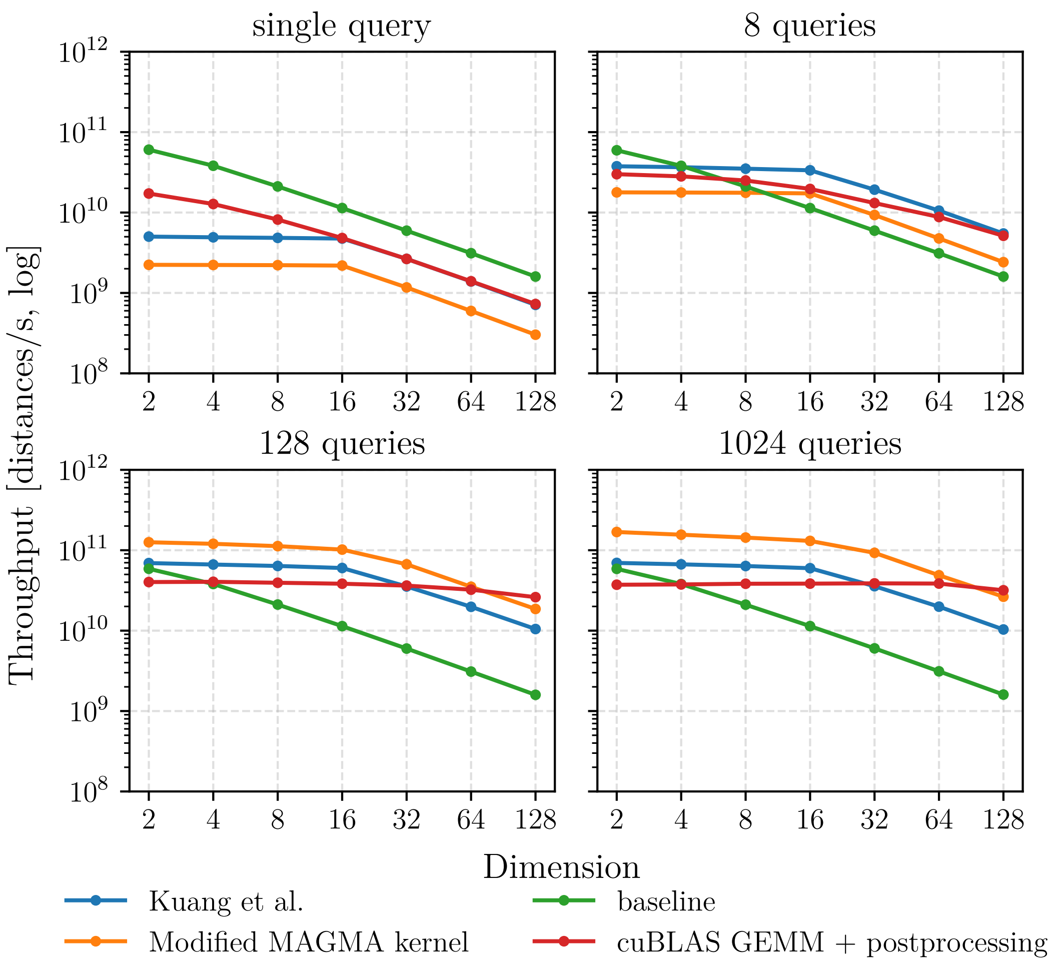

Figure 5.1: Throughput of squared Euclidean distance computation.

used on matrices of arbitrary size with user-defined operators, and we fixed some overflow errors with large matrices.

We use two different memory layouts for input matrices: column-major and row-major. Given n vectors with dimension d, matrices in the *row-major* layout store vectors consecutively in memory (i.e., all d elements of the first vector are stored at the lowest addresses, then elements of the second vector, etc.). The *column-major* layout is transposed row-major layout. The memory layout of input matrices depends on what is optimal for a given kernel. The baseline kernel and the modified MAGMA kernel have input matrices in the column-major layout. The kernel by Kuang et al. [31] expects both input matrices to be in a row-major layout. The cuBLAS matrix multiplication kernel was not affected significantly by the memory layout in our experiments. However, it is optimal to use the column-major layout for the matrix of database vectors in our postprocessing kernels. The distance matrix (output of the distance kernels) stores all distances from the first query at the lowest addresses, then distances from the second query, etc. This is an optimal layout for all k-selection kernels tested in this chapter.

Figure 5.1 shows the throughput (number of distances computed per second) of the kernels for different numbers of queries q. The y-axis is logarithmic. The size of the database n is chosen so that the amount of work ($n \times q \times d$ where d is the dimension of vectors) is constant for different numbers of queries q and dimension of vectors d. If the matrices would not fit into the memory of our GPU, we limit the database size (n) to a maximal admissible power of two.

Single-query instances do not benefit from caching as vector components, once loaded in an on-chip memory, are never reused. Therefore, it is difficult to beat the baseline kernel, which outperformed all other tested kernels for $q = 1$. As the number of queries increases, it becomes useful to cache some vectors to limit the number of global memory transactions. The kernel by Kuang et al. [31] outperformed other kernels for a small number of queries because it caches small parts of the input vectors in shared memory. The MAGMA kernel also caches parts of the input. However, the cached tiles are larger when compared to the previous kernel, which seems to be detrimental for a very small number of queries.

For a moderate number of queries (starting at $q = 128$), the modified MAGMA kernel outperformed all other kernels for small dimensions $d \leq 128$. The cuBLAS approach caught up with the MAGMA kernel at $d = 128$, outperforming the kernel for higher dimensions. However, Euclidean distance or the more general L_p distance is not a good distance function for high dimensional data [11, 46]. In practice, dimensionality reduction techniques [18], such as Random projection or Principal Component Analysis, are often used.

5.3 Selection optimizations

This section evaluates the effectiveness of k-selection optimizations proposed in Chapter 4. We evaluate Bitonic sort optimizations, the speed-up of buffering when compared with a partial sorting kernel, and global memory prefetching.

5.3.1 Bitonic sort

We evaluate Bitonic sort optimizations on a partial sorting kernel because it spends the majority of time sorting and merging. We empirically determined an optimal thread block size for this kernel to be 128 for $k \leq 512$ and 256 for larger values of k.

Figure 5.2 shows the effectiveness of Bitonic sort optimizations. The baseline kernel implements Bitonic sort in shared memory without warp shuffles. Warp shuffle instructions helped for a small k, but its effectiveness degrades with growing k because the portion of Bitonic sort steps implemented with warp shuffles is decreasing. Our final implementation of Bitonic sort in registers achieved a consistent speed-up of about two for all tested values of k.

5.3.2 Buffering

We added buffering to the previous partial sorting kernel (Section 4.1). Figure 5.3 shows the speed-up of buffered k-selection when compared with our implementation of the partial Bitonic sort. Buffering improved the performance of the kernel significantly as the most computationally intensive operation (sorting and merging the buffer) is significantly less common (Section 4.1.6).

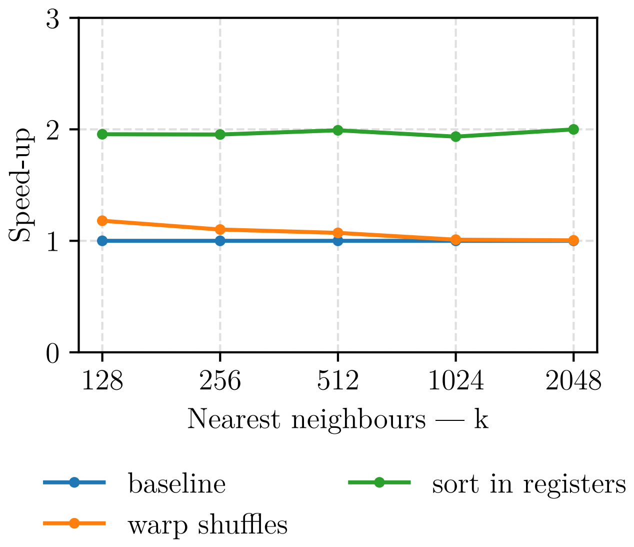

Figure 5.2: Bitonic sort optimizations effectiveness.

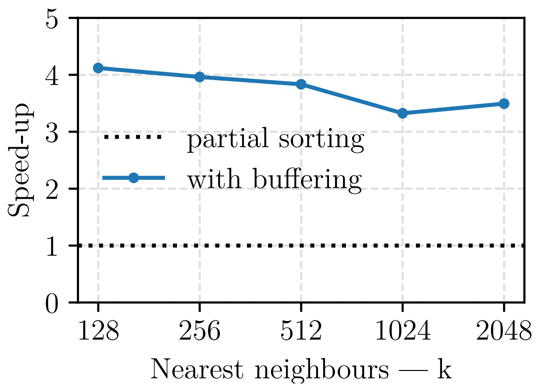

Figure 5.3: Speed-up of buffering in the memory optimized version of partial Bitonic sort.

There are instances in which this optimization would not perform as well. For example, if the list of distances was sorted in descending order, the kernel would degenerate to partial sorting (Section 3.5) as it would have to sort and merge every segment of k consecutive distances. We tested buffering on a uniformly random distribution of vectors in a d-dimensional cube. However, buffering needs a randomly shuffled list of distances to be effective, which is a less strict assumption.

5.3.3 Global memory throughput

We keep enough global memory transactions in flight by starting several global memory reads per thread in each iteration and using prefetch instructions (Section 4.1.5). Figure 5.4 shows speed-up compared to a kernel which reads only one value per thread. The kernel with prefetching is identical to the kernel without prefetching, except it inserts prefetch instructions in addition to multiple reads per thread. The prefetching kernel thus starts twice as many reads. However, half of the reads only store the values in the $L2$ cache, which might be evicted from the cache before the kernel processes the values.

Multiple reads per thread notably improved the throughput of our kernel in all cases, even for large $k = 2048$. We saw a speed-up above two in some instances. Register pressure turned out to be less of a problem than we expected. Inserting prefetch instructions in addition to multiple items per thread slightly improved the computation time. However, the difference gets smaller as we increase the number of items per thread.

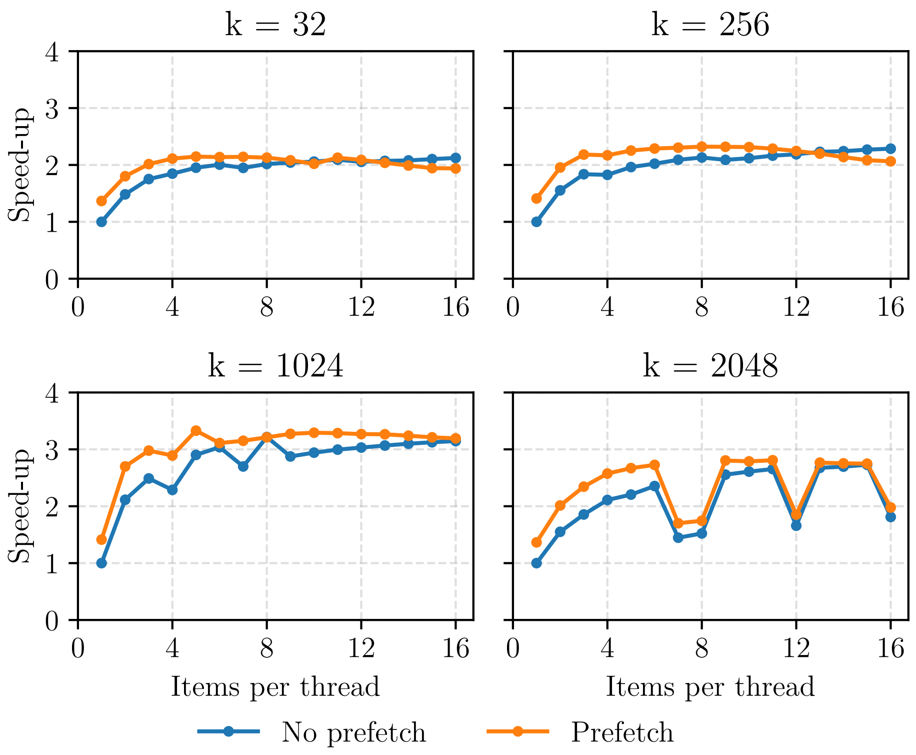

Figure 5.4: Speed-up of prefetching and multiple reads per thread.

5.4 Single-pass selection

In this section, we evaluate single-pass k-selection kernels. Single-pass kernels read all distances from global memory only once. These solutions use on-chip memory (registers, shared memory, or $L1$ cache) for temporary storage for the computation. The parameter k is expected to be fairly small ($k \leq 2048$) in order for auxiliary data structures to fit into an on-chip memory. However, this requirement is not a limiting factor for most kNN applications.

5.4.1 Multi-query selection

We begin by evaluating multi-query selection because these solutions can effectively leverage all cores of a GPU. However, algorithms evaluated here can be used to solve single-query problems using the approach discussed in Section 3.6. We will evaluate the effectiveness of this approach later.

The input of all kernels is a $q \times n$ distance matrix (32-bit IEEE 754 float) where q is the number of queries and n is the database size. The label of each database object is given implicitly as an index of the corresponding distance in the distance matrix. The expected output is two $q \times k$ matrices: top k distances and corresponding labels (32-bit integers) for all queries.

We evaluate the Warp select method (Section 3.6.3) from the FAISS library, buffered Merge queue (Section 3.6.2) from the fgknn library, and our Bitonic Select (bits) kernel (Section 4.1), which uses all threads of a thread block to evaluate each query. Additionally, we included the Block select method from the FAISS library, a variant of the Warp select method. It uses the whole thread block for each query such that each warp processes a subset of input data, and the partial results from all warps are merged at the end of the kernel. Straightforward adaptations of the serial kNN algorithm, such as the data-parallel approach (Section 3.6.1), performed significantly worse than other solutions, so they are not included in the results presented here.

We ran all kernels with varying database sizes from 32 thousand to one million objects. The number of queries was chosen so that the whole computation fits into our GPU's global memory, and the size of the distance matrix remains constant in all runs.

Warp select parameters

We noticed that the default configuration of the Warp select method (Section 3.6.3) from the FAISS library [6] could perform better on our GPU. We ran top k selection using the kernel with several thread queue sizes $2, \dots, 10$. For bigger thread queues, the register pressure becomes an issue (especially for large k).

Figure 5.5 shows the throughput of the Warp Select kernel with a database of one million vectors and two thousand queries. We only show the best configuration for each k for clarity. The numbers above each data point indicate the thread queue size used for that value of k. The default configuration is shown as a dashed line in the figure. The best alternative thread queue size from this test significantly improves the computation time for $64 \leq k \leq 128$.

Comparison

Figure 5.6 shows the throughput (number of distances processed per second) of the tested kernels. The second scale for the y-axis on the right shows throughput as a portion of a peak theoretical throughput of our GPU [1].

For clarity, we did not include the default configuration of WarpSelect and BlockSelect from the FAISS library. However, in all cases, our tuned parameters performed on par with or better than the default parameters.

Implementation of our bits kernel outperformed all other tested kernels in all configurations. The throughput of the bits kernel approaches 80% of peak theoretical throughput. It performed best on large databases with a relatively small number of queries $q \leq 2048$. This is coincidentally the configuration where we

[1]Tesla V100 has advertised memory bandwidth of 900 GiB/s [9].

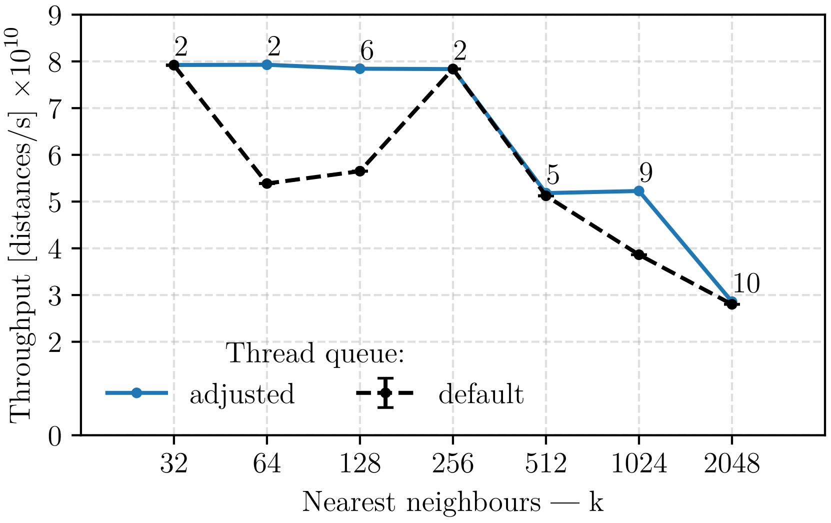

Figure 5.5: Parameter selection for Warp Select on Tesla V100. The numbers above the data points denote the best thread queue size for each k.

saw the best throughput for Euclidean distance computation using the modified MAGMA kernel (Figure 5.1).

5.4.2 Single-query selection

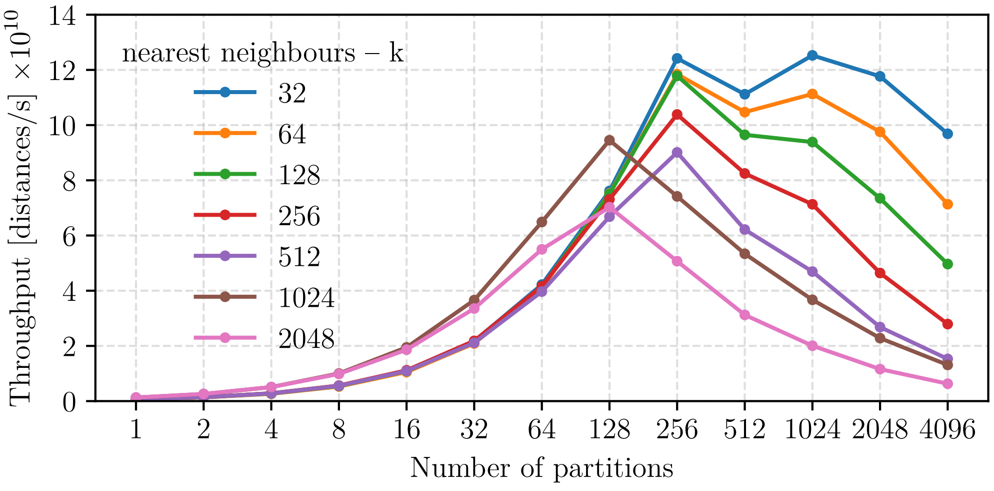

Figure 5.7: Evaluation of single-query bits kernel.

All kernels from the previous section would use only one thread block for single-query problems. This is suboptimal as most GPU cores would not be utilized. We adapted our multi-query bits kernel for single-query problems using the approach described in Section 3.6. The list of distances is split into multiple smaller partitions, which are assigned to different thread blocks. Once all thread blocks find their top k, the partial results are merged by one final pass over the partial results using the bits kernel. While this is not a single-pass kernel, we can achieve throughput similar to our multi-query bits kernel on large databases.

Figure 5.7 shows the throughput of the single-query bits kernel on a database with 128 million vectors with varying numbers of partitions. Choosing the correct number of partitions is crucial for performance as it scales only up to a certain point. The kernel had a peak throughput with 128 and 256 partitions for $k \geq 1024$ and $k \leq 512$, respectively, on Tesla V100.

5.4.3 Fused kernel

We compare the fused kernel (Section 4.2) with the best-performing multi-query k-selection kernel from the previous section (the bits kernel) and the best-performing distance kernel for small dimensions (the MAGMA kernel modified to compute L_2 distances, Section 5.2). The time to compute distances and the time to produce the top k result is added, and throughput is computed from this total time in the case of the two-stage approach.

Figure 5.8 shows the throughput (number of distances computed and processed per second) of our fused kernel and the two-stage approach with a database of half a million vectors and eight thousand queries. In the two-stage solution, the smallest k is $k = 32$ (the warp size). The dashed line in the figure extrapolates the result for $k = 32$ to smaller values of k. In the fused kernel, one thread block processes more than one query, so there is enough work for all threads, even if k is smaller than the warp size.

The fused kernel performed better than the two-stage solution for small dimensions $d \leq 32$ and especially for small $k \leq 64$. The two-stage solution is better for larger values of the parameters. However, the fused kernel requires less memory since it does not store the distance matrix. As a result, it can answer queries in larger databases, so it may be useful even in cases where it performed slightly worse than the two-stage approach.

An essential difference between the kernels is that the fused kernel requires more queries than the bits kernel to utilize all GPU cores fully. This is because the fused kernel processes several queries in one thread block, so it naturally creates fewer thread blocks than the bits kernel. Figure 5.9 compares the performance of the kernels with varying numbers of queries and database size. The dimension of vectors and the number of nearest neighbours is fixed at $d = 16$ and $k = 32$, respectively. The fused kernel achieved its peak performance with at least eight thousand queries.

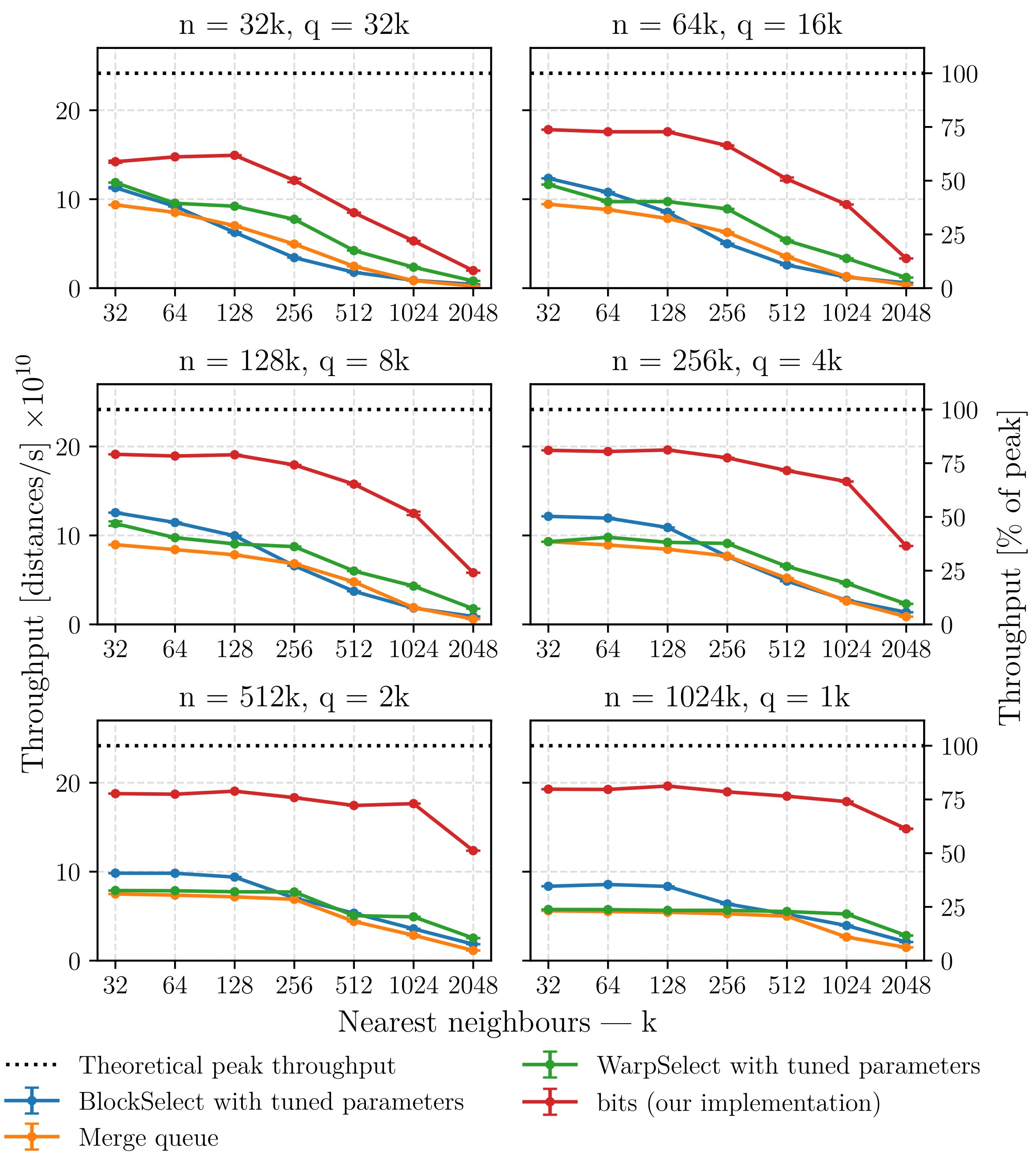

Figure 5.6: Evaluation of single-pass k-selection methods with q queries and n vectors in the database.

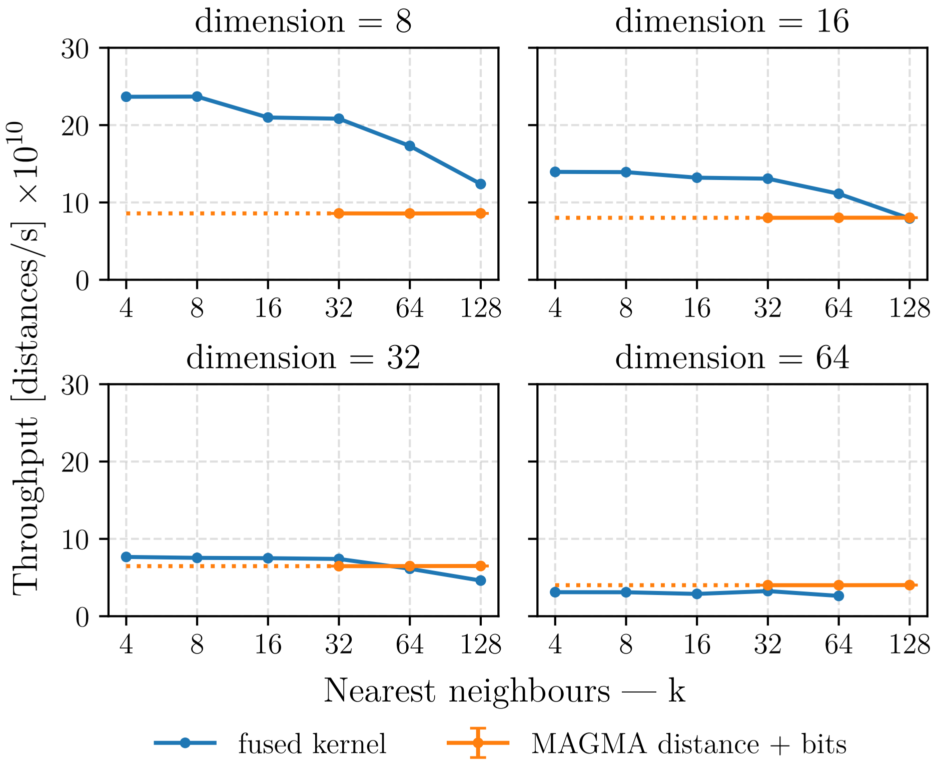

Figure 5.8: Comparison of the fused kNN kernel with bits and MAGMA kernel for distance computation.

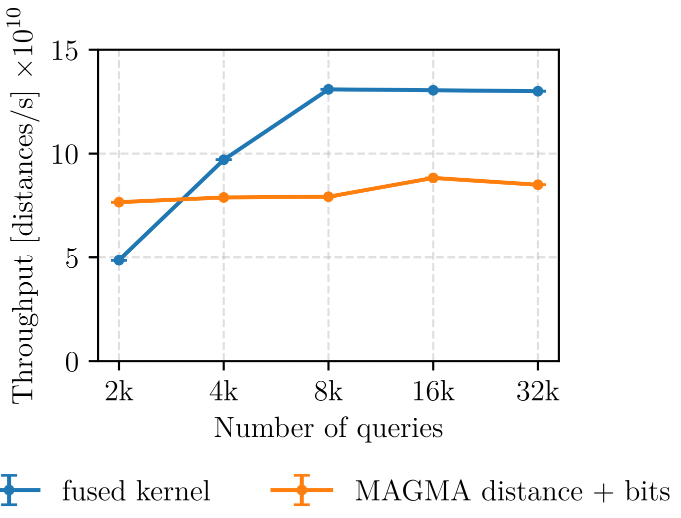

Figure 5.9: Throughput of the fused kNN kernel with $d = 16$, $k = 32$, and varying number of queries.

5.5 Multi-pass selection

Multi-pass kernels can read a distance from global memory several times. They are inherently slower than single-pass kernels, but they can find k nearest neighbours for much larger values of k. This category includes sorting kernels and kernels based on the Quick Select algorithm discussed in Section 4.3 (Sample Select and Radix Select).

5.5.1 Sample select

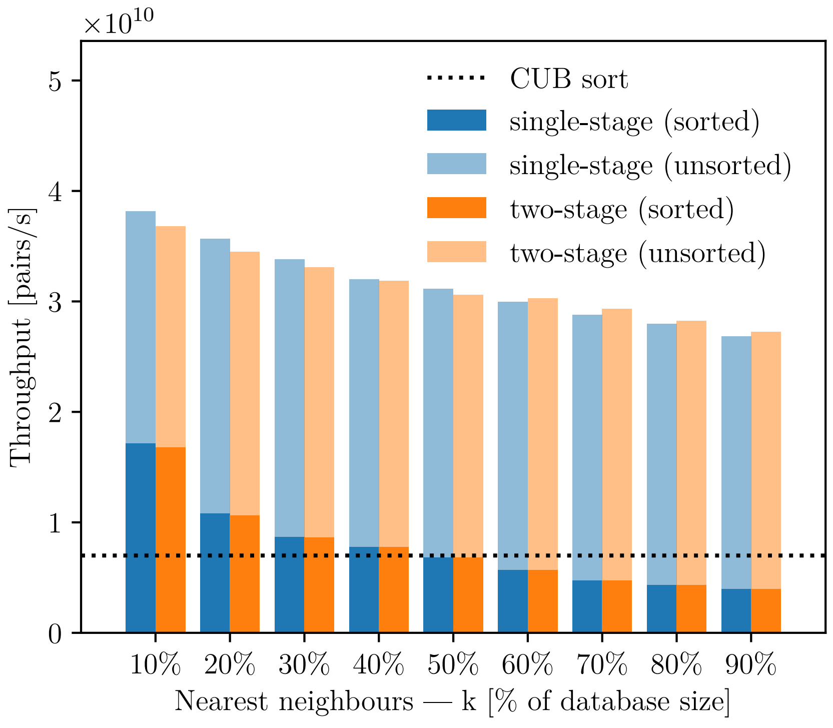

Figure 5.10: Comparison of different k-selection approaches in Sample Select with million objects in database and a thousand queries.

This section compares single-stage and two-stage implementations of the kNN method derived from the Sample Select algorithm (Section 4.3). Both methods receive two $q \times n$ matrices (n is the number of database vectors and q is the number of queries) — one with precomputed distances and one with object indices. They produce two $q \times k$ matrices — top k distances and corresponding indices. The output for each query is sorted using the segmented radix sort implementation from the CUB library [1].

As a baseline kNN implementation, we use radix sort from CUB to sort the distances. We use several CUDA streams[2] in the baseline to fully utilize all cores of our GPU when sorting distances from several independent queries. The CUB library also contains a segmented radix sort explicitly intended for sorting several arrays in parallel. However, it performed worse on a database of a million objects than using the single array implementation with CUDA streams.

[2]We empirically determined an optimal number of streams to be 8.

Figure 5.10 shows the throughput (number of distance and index pairs processed per second) of each tested configuration. We show both the throughput without sorting the top k result and with sorting. The throughput of the sorted kernels was better than the optimized CUB radix sort procedure (visualized as a dashed line in Figure 5.10) for $k \leq 30\%$ of the database. When k is more than half of the database, sorting the whole database is faster. The single-stage kernel was slightly faster than the two-stage kernel for small k, although the difference is negligible.

Both kernels performed significantly better than CUB radix sort if the output is not required to be sorted. However, it is important to note that in this case, the output for each query could be larger than k in case several distances are precisely equal to the kth smallest distance.

5.5.2 Partitioning algorithms

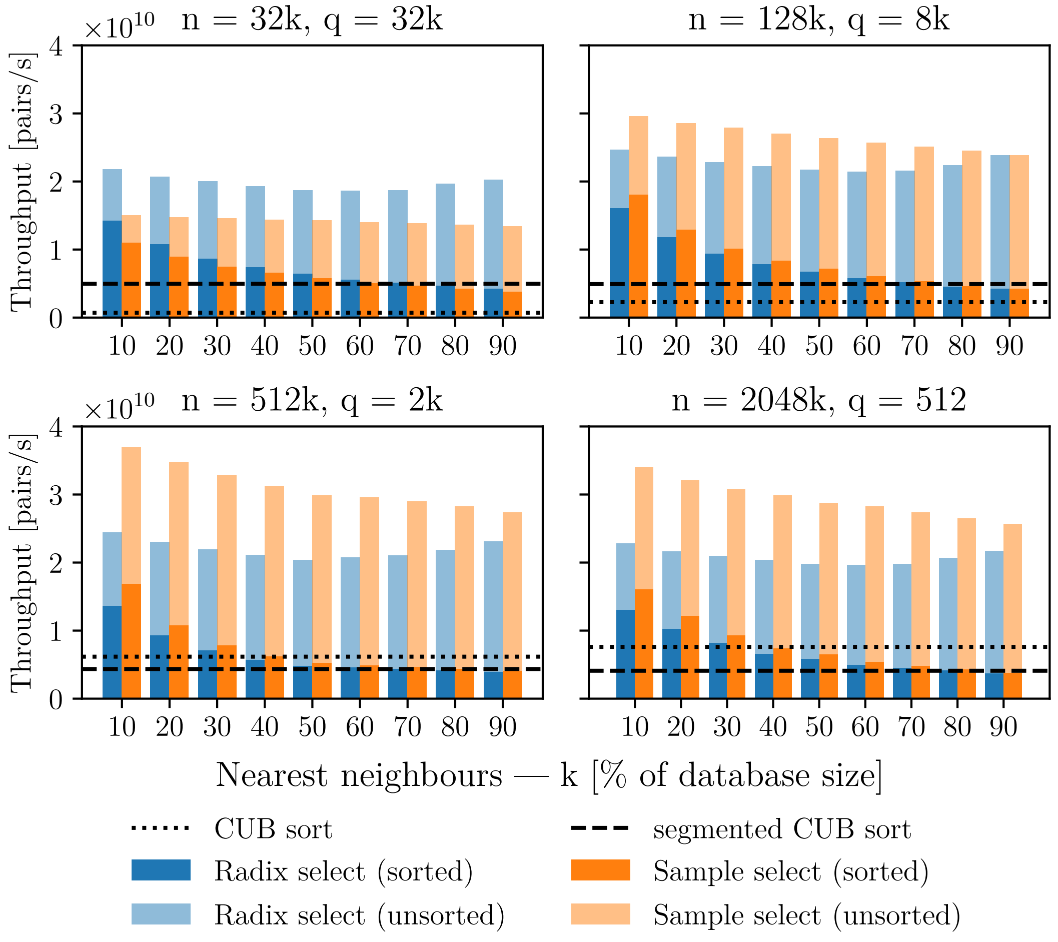

Figure 5.11: Evaluation of multi-pass k-selection methods.

Section 3.7 discusses different algorithms for partitioning distances which lead to two k-selection kernels: Sample Select and Radix Select (Section 3.7.2). In this section, we compare the kernels in various configurations. The Radix Select kernel is very similar to the Sample Select kernel, except it uses radix digits for partitioning distances to buckets. We also show the throughput of simply sorting the database using two optimized radix sort procedures from the CUB library [1]. The segmented CUB radix sort, which is specifically tailored to sort several arrays in parallel, and an ordinary single-array CUB radix sort implementation in multiple CUDA streams as described in the previous section.

Figure 5.11 shows our results for different sizes of the database n and the number of queries q. Sample Select and Radix Select performed significantly better than our baseline implementation if sorting the result is optional. For sorted top k results, the Sample Select method performed better than simply sorting the database for $k \leq 30\%$ of the database on large databases (a database with at least half a million vectors) and for $k \leq 50\%$ of the database on small databases (a database with less than a quarter of a million vectors). The Sample Select kernel performed better than Radix Select in all configurations except for a small database with only 32 thousand vectors.

5.6 Final kernel

Previous sections evaluated kernels for parts of the kNN problem in specific circumstances. We now use a broader approach that compares final kNN kernels using the best-performing kernels for distance computation and k-selection from this chapter.

Figure 5.12 shows the throughput of multi-query kNN kernels, including distance computation using the modified MAGMA kernel for squared Euclidean distances. The time to compute distances and time to select the top k results is added, and throughput is computed from this total time except in the case of the fused kernel. All kernels were given a database of 128 thousand vectors and 8 thousand queries. We compare the best-performing kernels from this chapter:

- the fused kernel (Section 4.2), which computes Euclidean distances and finds the top k in one kernel call.
- the Sample Select kernel (Section 4.3), an adaptation of the Quick Select algorithm for GPUs (produces an unsorted result).
- Our Bitonic Select (bits) kernel (Section 4.1).
- A variant of the bits kernel, which keeps the top k result in global memory, so it works for larger values of k (produces an unsorted result).

The rate at which small k-selection kernels process distances is roughly equal to the rate of computing the distances for small dimensions. However, as the dimension grows, the rate of producing distances drops below all k-selection kernels including the unsorted multi-pass Sample Select kernel (at $d = 128$).

We can summarize our results as follows. Given q queries, k nearest neighbours, dimension of vectors d, and size of the database n, the fused kernel outperformed other kernels if $d \leq 32, k \leq 64, q \geq 8000$. Otherwise, the bits kernel performed best for $k \leq 2048$ for multi-query and single-query problems. The bits kernel can be modified to store the top k result in global memory. This modification worked best if $k = 4096$ but it was quickly outperformed by the Sample

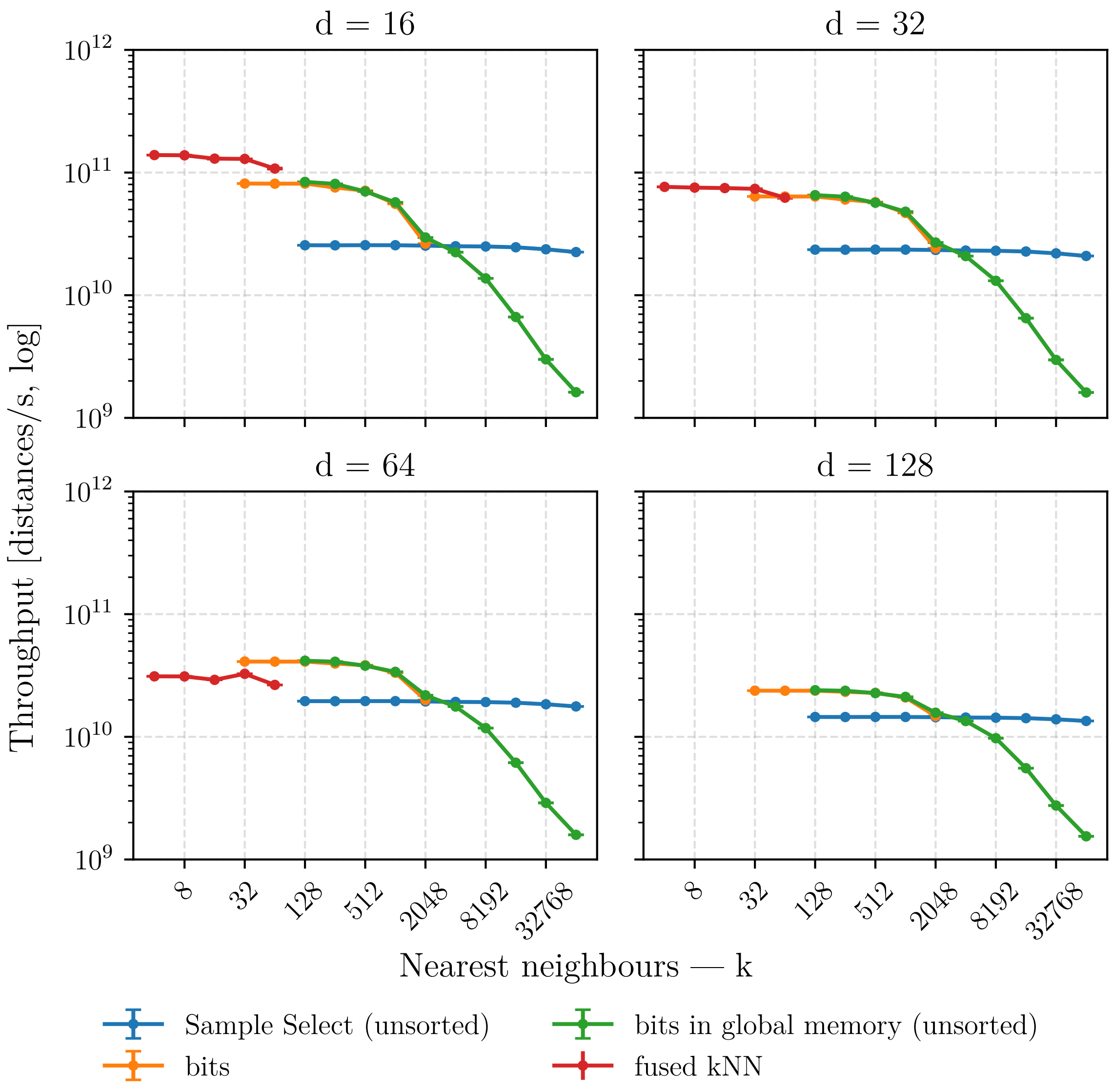

Figure 5.12: Throughput of multi-query kNN kernels.

Select kernel ($k \geq 4096$). However, the Sample Select kernel was faster than the Radix Select algorithm only in sufficiently large databases ($n \geq 100000$). The Radix select algorithm performed better in smaller databases.

5.6.1 Comparison with a CPU implementation

We implemented a parallel kNN kernel on a CPU. The kernel expresses distances in terms of a dot product, computed using a matrix multiplication algorithm (Section 3.3.1). We used the Eigen library [5] to compute the distances. The top k selection is implemented with a binary heap and parallelized using OpenMP [10]. We evaluated the kernel on a database with two million vectors and one thousand queries.

The speed-up of a GPU kNN implementation (modified matrix multiplication from the MAGMA library and our bits kernel) when compared with our parallel CPU implementation is shown in Figure 5.13. It does not factor in data transfers.

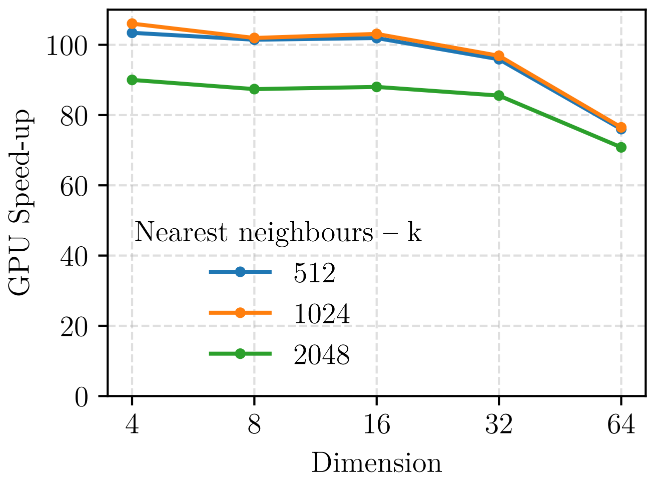

Figure 5.13: Speed-up of GPU kNN (MAGMA distance and the bits kernel) when compared with a parallel CPU implementation.

Figure 5.14 shows a ratio of the mean kNN computation time to the mean data transfer time. Data transfers include transferring the input to the GPU and transferring the output to the main memory of the CPU. Data transfers take a relatively short time for small dimensions, but they quickly become a limiting factor of the computation. However, in a typical configuration, the matrix of database vectors is the largest block of memory needed to be transferred to the GPU. If we use the same database to process multiple batches of queries, we only have to transfer the matrix once.

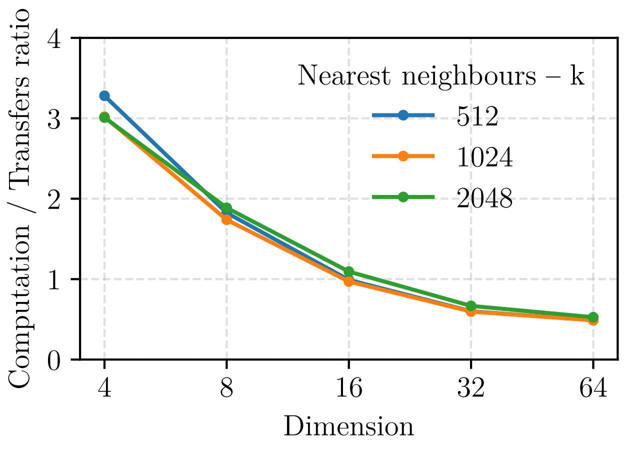

Figure 5.14: The ratio of the computation time to the data transfer time.

6. Conclusions

While a naive implementation of a k nearest neighbours algorithm on a CPU is straightforward, an efficient, parallel implementation on a GPU requires a customized solution to utilize hardware resources fully. Moreover, an optimal approach to kNN depends on the problem's parameters, like the database size, the number of nearest neighbours, and the used distance function, among others. In this work, we evaluated GPU implementations of kNN with various configurations. We evaluated both parts of the problem (distance computation and k-selection) separately and considered fusing them into one kernel.

No general approach to distance computation would work for all distance functions. However, the most common distance functions like Euclidean distance or cosine similarity reduce to matrix multiplication [32]. A modification of the matrix multiplication kernel from the MAGMA library [7] outperformed other distance kernels for multi-query kNN problems with a sufficient number of queries. Single-query kNN problems with a simple distance function like Euclidean distance do not benefit from complicated caching strategies as no vector component from the input vectors is used more than once.

We suggested some optimizations in several areas of the top k selection. In particular, we implemented an incremental, multi-query top k selection algorithm based on Bitonic sort [13], which we call bits. We also tried to fuse distance computation with top k selection. The fused kernel combines the distance computation approach proposed by Kruliš et al. [30] with our bits implementation. Lastly, we adapted the Sample Select [40] algorithm for kNN with a large value of k and compared several partitioning strategies.

Our implementation of the bits kernel reached up to 80% of peak theoretical throughput on Tesla V100 with a typical configuration (an extensive database, a relatively small number of queries, and $k \leq 128$). It outperformed other state-of-the-art multi-query k selection kernels in all tested configurations and can effectively find an answer for single-query problems with a slight modification. We also showed that the kNN problem can be implemented as a fused kernel, which outperformed our bits kernel in some configurations (small dimension $d \leq 32$ and $k \leq 32$ with sufficient queries $q \geq 8000$). The bits kernel and most GPU k-selection kernels in literature only work for small k ($k \leq 2048$). The modified Sample Select kernel can be used for larger values of k, and it outperformed a brute force solution (sorting the whole database) by a significant margin when the top k output does not have to be sorted. Even if we sort the top k result, the Sample Select method outperformed the brute force solution when k was at most 30% of the database size.

We looked at the exact kNN algorithm implementation on a GPU and found some room for improvement. In the future, a similar survey can be done for related problems like approximate kNN and parallel indexing methods for kNN to avoid the computation of costly distance functions.

The prototype implementation of all kernels presented in our work is publicly available[1] for reproduction of experiments and further use.

[1]https://gitlab.mff.cuni.cz/hanakdr/knn

Bibliography

[1] CUB. `https://nvlabs.github.io/cub/`. Accessed: 2021-08-02.

[2] cuBLAS. `https://docs.nvidia.com/cuda/cublas/index.html`. Accessed: 2021-08-02.

[3] CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2021-08-08.

[4] cuRAND. `https://docs.nvidia.com/cuda/curand/host-api-overview.html`. Accessed: 2022-04-30.

[5] Eigen. `https://eigen.tuxfamily.org/`. Accessed: 2023-06-15.

[6] FAISS. `https://github.com/facebookresearch/faiss`. Accessed: 2021-08-08.

[7] MAGMA. `https://icl.cs.utk.edu/projectsfiles/magma/doxygen/routines.html`. Accessed: 2021-08-02.

[8] moderngpu 2.0. `https://github.com/moderngpu/moderngpu`. Accessed: 2021-08-08.

[9] NVIDIA Tesla V100. `https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf`. Accessed: 2022-05-27.

[10] OpenMP. `https://www.openmp.org/`. Accessed: 2023-06-15.

[11] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.

[12] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–1, 2012.

[13] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[14] S. Baxter. moderngpu 2.0. `https://github.com/moderngpu/moderngpu/wiki`, 2016.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[16] D. Bednárek, M. Brabec, and M. Kruliš. Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64:175–193, 2017.

[17] C. Beecks, M. S. Uysal, and T. Seidl. Signature quadratic form distances for content-based similarity. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 697–700, 2009.

[18] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.

[19] M. Boston et al. A dynamic index structure for spatial searching. In *Proceedings of the ACM-SIGMOD*, pages 547–557, 1984.

[20] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[21] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, volume 97, pages 426–435. Citeseer, 1997.

[22] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008.

[23] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

[24] O. Green, R. McColl, and D. A. Bader. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340, 2012.

[25] M. Herf. Radix tricks. `http://stereopsis.com/radix.html`, 2001. Accessed: 2022-04-21.

[26] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[27] M. Kruliš, S. Kirchhoff, and J. Yaghob. Perils of combining parallel distance computations with metric and ptolemaic indexing in knn queries. In *International Conference on Similarity Search and Applications*, pages 127–138. Springer, 2014.

[28] M. Kruliš, J. Lokoč, C. Beecks, T. Skopal, and T. Seidl. Processing the signature quadratic form distance on many-core gpu architectures. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2373–2376, 2011.

[29] M. Kruliš, H. Osipyan, and S. Marchand-Maillet. Optimizing sorting and top-k selection steps in permutation based indexing on gpus. In *East European Conference on Advances in Databases and Information Systems*, pages 305–317. Springer, 2015.

[30] M. Kruliš and M. Kratochvíl. Detailed Analysis and Optimization of CUDA K-Means Algorithm. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Q. Kuang and L. Zhao. A practical GPU based kNN algorithm. In *Proceedings. The 2009 International Symposium on Computer Science and Computational Technology (ISCSCI 2009)*, page 151. Citeseer, 2009.

[32] S. Li and N. Amenta. Brute-force k-nearest neighbors search on the GPU. In *International Conference on Similarity Search and Applications*, pages 259–270. Springer, 2015.

[33] M. Mareš. *Pruvodce labyrintem algoritmu.* CZ. NIC, zspo, 2021.

[34] M. L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, 1994.

[35] F. Moreno-Seco, L. Micó, and J. Oncina. Extending LAESA fast nearest neighbour algorithm to find the k nearest neighbours. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 718–724. Springer, 2002.

[36] N. Nodarakis, A. Rapti, S. Sioutas, A. K. Tsakalidis, D. Tsolis, G. Tzimas, and Y. Panagis. (A) kNN Query Processing on the Cloud: A Survey. In *International Workshop of Algorithmic Aspects of Cloud Computing*, pages 26–40. Springer, 2016.

[37] C. J. Nolet, D. Gala, E. Raff, J. Eaton, B. Rees, J. Zedlewski, and T. Oates. Semiring Primitives for Sparse Neighborhood Methods on the GPU. *arXiv preprint arXiv:2104.06357*, 2021.

[38] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path-parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1611–1618. IEEE, 2012.

[39] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pages 46–65. Springer, 2003.

[40] T. Ribizel and H. Anzt. Approximate and exact selection on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 471–478. IEEE, 2019.

[41] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[42] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[43] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. Technical report, 1987.

[44] X. Tang, Z. Huang, D. Eyers, S. Mills, and M. Guo. Efficient selection algorithm for fast k-nn search on gpus. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 397–406. IEEE, 2015.

[45] P. Terdiman. Radix Sort Revisited. `http://codercorner.com/RadixSortRevisited.htm`, 2000. Accessed: 2022-04-21.

[46] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

A. Attachments

The attachements contain a prototype implementation of all kernels presented in our work. The impementation is also available online [1].

Prerequisites include `cmake` version at least 3.17, C++ compiler that supports C++20, and CUDA. The following commands download dependencies and compile all kernels from our work:

1. `git submodule update --init --recursive`
2. `mkdir build-release`
3. `cmake -B build-release -DCMAKE_BUILD_TYPE=Release -DCMAKE_CUDA_ARCHITECTURES=70 .`
4. `make -C build-release`

The build process creates an executable program `knn`, which can run our benchmarks. A description of implemented benchmarks can be found in `README.md`. We assume a Volta architecture. A different value of `CMAKE_CUDA_ARCHITECTURES` has to be used for other architectures.

A.1 Structure

- `src/distance` includes implementation of distance kernels.
- `src/topk/singlepass` includes implementation of single-pass k-selection kernels and the fused kernel.
- `src/topk/multipass` includes implementation of multi-pass k-selection kernels.
- `doc/05-evaluation` contains bash scripts for running all experiments (`run-*.sh`). All bash scripts contain a SLURM configuration to run the experiments in KSI clusters.

[1] https://gitlab.mff.cuni.cz/hanakdr/knn