AliceVision: GPU Programming for Depth Estimation

Feature descriptor matching with Locality Sensitive Hashing optimized using OpenACC

Hilmar Røtnes Widerberg

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

AliceVision: GPU Programming for Depth Estimation

Feature descriptor matching with Locality Sensitive Hashing optimized using OpenACC

Hilmar Røtnes Widerberg

AliceVision: GPU Programming for Depth Estimation

Hilmar Røtnes Widerberg

May 16, 2022

Abstract

The scale-invariant feature transform (SIFT) algorithm is an algorithm for matching distinctive points of interest between pictures of the same item but from different angles. The algorithm came out in a paper titled "Distinctive Image Features from Scale-Invariant Keypoints" published by David Lowe in 2004 [9]. The algorithm is used for computer vision, and there exists many different implementations of it.

The SIFT algorithm can be used for depth estimation, that is finding out how far away objects are, typically so one can figure out what objects should be in front of other objects.

Depth estimation can in turn be used for Mixed Reality applications. One of the goals of Alice Vision is to achieve Mixed Reality results in real-time so that the results of a dataset can be shown as it's being worked with, the problem is of course the computation time.

A major bottleneck in depth estimation is in trying to match SIFT feature descriptors (each describing the local area around that point). There is one feature descriptor for every point of interest in the image, and each feature descriptor consists of 128 floating point values. When comparing two images, every feature descriptor in the first image needs to be matched (or attempt to be matched) with a feature descriptor in the other image. In order to match a feature descriptor with another feature descriptor, one needs to make sure that there is no better match for the feature descriptor, for a naive nearest neighbor algorithm this means we have to compare our feature descriptor to every feature descriptor in the other image.

Needless to say, for large and feature rich images with many points of interest, this computation can be very slow. The idea of this master's thesis is to implement an alternative algorithm to find nearest neighbors using a Locality Sensitive Hashing (LSH) approach, then optimizing implementations of this algorithm using the OpenACC GPGPU[1] programming interface.

The goal of this thesis is to create code that can potentially be used down the line in an already existing implementation of SIFT called PopSift [3].

Since the programs created are optimized using OpenACC, a very general GPGPU programming interface, the program will not be geared towards any specific CPU or GPU architecture. However, as I will point

[1]GPGPU stands for General-Purpose Graphical Processing Unit, I will sometimes use the terms GPGPU and GPU interchangeably, but to be precise interfaces such as CUDA is made to program GPGPUs. Most modern GPUs are GPGPUs.

out in Section 4.1, the program will be tested on an "Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz" CPU and a "NVIDIA RTX2080Ti" GPU.

After the short chapter covering the various research questions relevant to this thesis, the thesis will start introducing various subjects. Chapter 2 will cover GPGPU programming in general, as well as two different GPGPU programming interfaces: CUDA and OpenACC. The CUDA programming interface will be introduced in order to gain important insight into how GPGPUs are programmed. CUDA is a lower level interface than OpenACC, and therefore gives a more in-depth view into the inner workings of GPGPUs. OpenACC will then be introduced as a simpler, more general, GPGPU programming interface. One can program OpenACC without knowing how CUDA works, but knowledge of CUDA leads to a deeper understanding as to how OpenACC directives actually translate serial code into code that can run on GPGPUs.

Chapter 3 will be about feature descriptor matching. This chapter starts out with a section covering how the features are created using the SIFT algorithm. The next section covers Nearest Neighbor algorithms used to match multi-dimensional vectors. The last section in this chapter is both the longest and perhaps most important introductory section, it covers the Locality Sensitive Hashing algorithm, an Approximate Nearest Neighbor (ANN) algorithm which will be used to match SIFT feature descriptors and optimized using OpenACC in this thesis.

After introducing the LSH algorithm, the next couple of chapters cover two different versions of the LSH algorithm, how they can be implemented, how they can be optimized, and how well they perform.

At the end of the thesis there will be a conclusion chapter with some final remarks, analyses, and suggestions of further research.

Contents

Part I

Introduction

Ordering and Relation of Introductory Sections

Within this introductory part of the thesis I will first present the research questions that this thesis seeks to address.

After having written about the research questions I will introduce two programming models for GPGPU computing [2], then I will introduce the problem at hand that is feature descriptor matching.

I will present the topics in this order as it is good to be familiar with the concepts of GPGPU computing before diving into the problem we are trying to solve. This should help the reader bear those concepts in mind while being presented with the problem we're trying to optimize.

Another advantage of presenting the topics in this order is that the section on feature descriptor matching naturally leads to the main problem of this thesis. If the topics were presented in the opposite order a large chapter about GPGPU computing could be perceived as a major detour.

Towards the end of the section on feature descriptor matching I will cover Locality Sensitive Hashing (LSH), an algorithm for efficiently matching features in a large feature space.

[2]GPGPU computing is covered in Chapter 2

Chapter 1

Research Questions

Before I begin introducing the various subjects that this thesis covers I will first present the various research questions guiding this thesis.

Research questions regarding OpenACC:

1. How suited is OpenACC to accelerate a Locality Sensitive Hashing (LSH) algorithm, and how well is it able to do so? What challenges occur when using OpenACC? Does the OpenACC interface have any limitations?

2. How does OpenACC deal with more complicated program behavior, such as the case where loop iterations are only semi-independent? How does one best utilize data movement in OpenACC?

Research questions regarding Locality Sensitive Hashing (LSH):

1. What are the different ways one can implement LSH? Which portions of LSH are best parallelizable, and are they parallelizable using GPGPUs?

2. What are the advantages of LSH over other nearest neighbor or approximate nearest neighbor algorithms?

3. Are there any preprocessing steps that could be taken to make LSH more effective? LSH uses hyperplanes to partition the search space. What are good types of hyperplanes, and how can they be generated?

4. Each LSH table adds to the time and resources that are required by the algorithm, what are different ways to reduce the number of tables needed? Semi-related (as it turns out): LSH does not guarantee a match within any range, is there any way to remedy this?

5. Is LSH useful for matching SIFT descriptors for the purpose of depth estimation? If not, can LSH still be useful for other purposes?

Chapter 2

GPGPU Computing

2.1 Introduction

Ever since the development of increasingly faster single-core architectures started to slow down there has been more and more focus on multi-core solutions. The concept of GPGPUs, together with the families of GPGPU architectures, is one of these multi-core solutions.

"GPU" stands for "Graphical Processing Unit" and GPUs were originally developed specifically to handle computer graphics. Computer graphics problems typically involve performing many similar, but largely independent tasks that can be done efficiently in parallel, and GPUs were made to optimize this. Coincidentally, the multi-core architecture designed specifically to more efficiently solve problems in computer graphics turns out to also be effective at solving other more general types of problems. After this became apparent, GPU manufacturers started working on designs and interfaces directed more towards general purpose programming giving us "General Purpose Graphical Processing Units" or "GPGPUs" for short.

Since most modern GPUs are also GPGPUs, I will henceforth often refer to "GPGPUs" as simply "GPUs", since this is shorter and more convenient.

2.1.1 CPU Parallelism

CPUs have themselves evolved to further exploit parallelism over the years. Some forms of parallelism can be seen as kind of improving latency depending on how one looks at it[1], this is the case for pipelining and Single

[1]The difference between a latency optimization and a throughput optimization can be a bit blurred in the cases where very fine-grained parallelism is used. By the strictest definition of latency, both pipelining and SIMD instructions are viewed as increasing throughput and not shortening the latency since they don't shorten the time of a single instruction but rather increase speed by performing multiple instructions or operations at once; by this definition, almost no form of parallelism can increase latency. This definition of latency is a bit too strict for my purposes here, and seems more relevant if one is speaking about CPU microarchitecture or some other similar subject. The reason I refer to them as kind of almost improving latency is that they can only perform a small number of operations at a time and are often useful to optimize serial programs; pipelining in particular improves performance of what is considered completely serial programs.

Instruction Multiple Data (SIMD) instructions, which use parallelization but only operate on a very limited scale. Modern CPUs are also divided into multiple cores used to increase throughput and to better facilitate multitasking.

Most computer programs are single threaded, so naturally these are the types of programs that the CPUs are made to optimize. We say that the CPU uses a latency-oriented design, meaning it tries to minimize latency by making single threaded programs run as fast as possible. CPU cores use all kinds of sophisticated techniques to minimize latency: they maintain a high clock frequency; they use branch prediction, instruction reordering, and data forwarding for more effective pipelining and less waiting time for data; they also use large caches to avoid cache misses.

CPUs also use data prefetching to fetch data before it is needed in the program, putting it into cached memory much closer to the processor. This way, CPUs manage to hide the latency by doing other useful work while waiting for data.

As a result of their latency oriented design, modern CPUs typically use most of their surface area for control and caching, and dedicate only a small amount of their surface area to actual computation using Arithmetic Logic Units (ALUs).

2.1.2 GPU Parallelism

GPUs take a completely different approach. While CPUs are designed to minimize latency, GPUs are designed to maximize throughput (i.e. they use a throughput oriented design). A GPU splits computation into many semi-independent threads, each running on their own small and semi-independent core. The cores are grouped together into larger groups where each group cooperates on running the same program in a Single Instruction Multiple Threads (SIMT) manner. SIMT was originally only a user-friendly abstraction on top of SIMD, but the latest architectures have added new features such as a program counter for every SIMT thread, making SIMT now somewhat distinct from SIMD not just from a programming perspective, but also from a hardware perspective.

While CPUs use most of their on-chip surface area for control and caching, GPUs instead dedicate most of their on-chip surface area to ALUs. Though memory accesses can be slower due to smaller caches, GPUs try to hide this latency by coalescing memory accesses[2], and also doing work while waiting for data to arrive; if the GPU does not have enough work to do in the meantime however, the latency becomes noticeable.

GPUs dedicate fewer resources to minimize latency than CPUs, they also maintain a lower clock frequency. Still, thanks to the incredible amount of cores available to them, GPUs are able to massively outperform CPUs for parallel tasks in which throughput is the bottleneck. On the flip-side, CPUs

[2]The GPU "coalescing memory accesses" means it groups sequential (or mostly sequential) memory accesses of mostly contiguous memory areas together and fetches them simultaneously instead of fetching them one by one.

massively outperform GPUs on serial tasks in which latency becomes the bottleneck.

The terminology here can be confusing as performing tasks in parallel, thus increasing throughput, also reduces the latency of the entire program. When we say that CPUs are designed to minimize latency, we generally mean they are designed to minimize the average latency of every instruction. In other words, CPUs are designed to optimize programs that are inherently serial in nature, incidentally programs for which GPUs perform very poorly. We can therefore say that CPUs prioritize minimizing *real time interaction* latency (at least to some degree), whereas GPUs largely ignore that type of latency and instead prioritize minimizing *completion time latency*, which it may achieve by increasing throughput for a throughput constrained problem.

Besides running a handful of fully independent threads, the other way CPUs use parallelism is through fine-grained parallelism such as pipelining and single threaded SIMD. This increases the throughput of only a few instructions at a time, so the effects are similar to optimizing the latency of every instruction, however they do this by technically focusing on throughput within a very fine-grained context. Since these optimizations work well also for more serial programs I referred to them as latency optimizations earlier.

2.1.3 Granularity

It should be mentioned that, due to the SIMT model as well as other factors, the cores used by a GPU employ a somewhat fine-grained form of parallelism compared to the cores used by a CPU. So one cannot, or at least should not, run many individual, single-threaded programs each on their own GPU core; this is what multi-core CPUs are for. Instead, GPUs are typically used for cases where you have one large parallelizable task, and where every thread will, most of the time with some exceptions, perform the same instructions; i.e. the threads run the same program on separate data.

On the other hand, when compared to single-threaded SIMD instructions the way they are deployed on CPUs, the parallelism on GPUs is more coarse-grained. It would therefore be a bad idea to start offloading work to a GPU for performing only a few operations on relatively small amounts of data, the overhead involved in parallelization would quickly overtake the amount of parallelizable work.

On the spectrum from coarse-grained to fine-grained parallelism, the GPU programming model lies somewhere in the middle.

2.2 GPGPU Programming Models

There are many different GPGPU programming models, interfaces and standards out there. In this project we are only going to be discussing two of them: CUDA and OpenACC.

2.2.1 Brief Introductions

CUDA is a GPGPU programming model created by NVIDIA. It is specially designed to program GPUs produced by NVIDIA, and because of this it will not work for GPUs provided by other manufacturers.

CUDA offers a lower level programming experience than OpenACC. It introduces a few extra programming paradigms to the language it is working with, along with an extensive API which the programmer can use to get a program running on a suitable NVIDIA GPU.

CUDA is typically used with C/C++, but also works with Fortran. It is compiled with a special compiler called nvcc. Programming in CUDA involves low level practices such as explicitly designing GPU kernel functions, and the programmer directly controlling the use of GPU resources such as shared memory.

OpenACC on the other hand is a more general, cross-platform, parallel programming standard developed by Cray, NVIDIA, and PGI [3]. It is used to program a multitude of hardware accelerators, produced by a variety of different companies.

OpenACC is a high level interface consisting of a number of compiler directives that can be added to programs written in one of its supported languages. OpenACC directives are added to the program as comments in Fortran, or as so-called pragmas in C. It offers portability and ease of programming as its main features.

At the time of writing OpenACC can only be used with C/C++ or Fortran, other languages can only use it indirectly through libraries written in one of these languages. Many different compilers support OpenACC; those that don't are free to ignore the compiler directives, and compile the program as if it was a serial program instead.

2.2.2 How CUDA and OpenACC Be Presented

There will first be a section on CUDA since knowledge about CUDA will give valuable context regarding how OpenACC works behind the scenes. Though CUDA is created specifically for NVIDIA GPUs, a lot of the abstract concepts carry over to other GPGPU programming models such as OpenCL; the semantics are different, but the overall concepts remain the same.

After having presented the CUDA programming model and some of its features, I will start a new section where I turn our attention to OpenACC. While describing OpenACC I will compare it with CUDA, and

[3]PGI was acquired by NVIDIA in 2013

I will sometimes draw parallels between the two interfaces when covering OpenACC's features.

In this thesis I will only present how these programming models work in C.

2.3 CUDA

This section is mostly based on the CUDA programming guide on NVIDIA's web page, as well as the book "Programming Massively Parallel Processors Programming Massively Parallel Processors: A Hands-on Approach" by David Kirk and Wen-mei Hwu [8].

CUDA (Compute Unified Device Architecture) is a GPGPU programming model developed by NVIDIA and used to program NVIDIA GPUs. CUDA is not a language in and of itself but rather an extension to other languages such as C/C++, the CUDA variant of which is called CUDA C. CUDA programs generally use the ".cu" extension and are compiled using the `nvcc` compiler. A CUDA C program is written with normal C/C++ code, the only parts of a C program that needs to change are parts of the program you wish to parallelize using CUDA concepts.[4]

2.3.1 Programming Model

In the parlance of CUDA the CPU is called the *host*, and the GPU is called the *device*. CUDA distinguishes between CPU memory and GPU memory, which are called host memory and device memory respectively. Traditionally, the CPU and GPU memories are separate, and memory on the CPU has to be transferred through a dedicated PCI-express bus before it can be used by the GPU and vice versa; transferring the memory is here the job of the programmer. Alternatively, we also have the unified memory model in which the CPU and GPU are able to seemingly use the same memory as long as they don't try to access it at the same time. Some architectures are better adapted to using unified memory than others; that is, on many laptop designs as well as on the motherboards of a few supercomputers, the CPU and GPU share memory, which provides true unified memory.

CUDA gives the programmer the ability to create parallel functions that, when called with some additional parameters passed through special syntax specifying the number and organization of threads, start to execute in parallel on the GPU using those provided threads. This type of function is called a *global function* or a *kernel*, and starting the parallel execution of that function on the aforementioned provided threads is referred to as *launching* the kernel. Because global functions execute on the GPU they have to be written in a special way in order to be effective.

CUDA divides the computation of the kernel into *blocks* which are further divided into *threads*; the blocks themselves are part of a greater *grid* of blocks. The grid and block dimensions directly determine the amount of blocks and threads that will run on kernel launch. The decision of what grid and block dimensions to use and consequently how many blocks and threads should be launched is up to the program that launches the kernel.

[4]Parts of the program heavily related to the parts one wishes to parallelize sometimes also have to be changed. Sometimes the program must be redesigned slightly in order to make it more parallelizable, though this has little to do with CUDA itself and more to do with parallelization of programs in general.

The blocks are executed on a Streaming Multiprocessor (SM) each of which can run a handful of blocks at a time. The number of blocks that can be run on a single SM depends on both a predefined maximum value as well as the size of the blocks (since the SM is also restricted in terms of the total number of threads it can run concurrently). Threads run on so-called "CUDA Cores", each SM will contain a large number of these CUDA cores.

Cuda threads cannot run fully independently at the same time. In today's GPUs they are grouped together into groups of 32 threads called warps; each warp will run the same instruction at the same time for all its threads. One warp can only run a single instruction at the same time, however it can perform this instruction for all its threads; threads can of course branch away from each other even if they are in the same warp, but in this case the threads will have to wait for each other. The fact that warps can run only one instruction at the same time, and that this instruction is run by multiple threads that each operate on one piece of data is why CUDA is often referred to as a SIMT (Single Instruction Multiple Threads) programming model. SIMT is similar to regular SIMD, but it achieves SIMD functionality through the use of threads [5] , which is distinctly different from how CPUs achieve SIMD functionality and thus it is given its own name. In newer architectures, a warp can run the same instruction for all threads even when the threads are not in the same place in the program. Since threads are grouped into warps it is generally a good idea, when launching a kernel, to specify the number of threads per block to be a multiple of the warp size, which is 32 for most current NVIDIA GPU architectures.

2.3.2 Memory Hierarchy

In order to program efficiently in CUDA one needs to be aware of the CUDA memory model. I will present the different types of memory in the order from fastest to slowest in the beginning, however the last three items have roughly equal performance:

1. Register memory is accessible within a single clock cycle and is the fastest type of memory, however there are only a few registers for every thread. It is important for efficiency to not use up more than the available register space since this will cause the extra memory to *spill* into *local memory*. Registers are individual to each thread.

2. Shared memory is the second fastest type of memory, almost as fast as register memory. This memory is shared between threads in a single block. Just like register memory, shared memory is stored on-chip.

3. Texture and constant memory are stored outside the chip, but performance is vastly improved by caching. Which one of these is

[5]Originally, the SIMT model only existed from a programmer's perspective; it was simply an abstraction achieved by the nvcc compiler and the JIT compiler in the device driver. For modern hardware however it is actually distinctly different from SIMD with the introduction of per-thread program counters.

faster depends on the details of the particular implementation. These types of memories are read-only. Since these types of memories are stored outside the chip they will be significantly slower than register and shared memory.

4. Global memory used to be the slowest type of memory together with "local" memory, as it didn't have an L1 cache. For modern architectures however, global memory does make better use of caching, and its performance is now comparable to that of texture and constant memory.

5. Local memory is only called *local* because it is individual to each thread. Despite its name, local memory is actually just global memory, but reserved by the compiler for an individual thread. Local memory is therefore, just like global memory, incredibly slow compared to register and shared memory. Newer architectures sometimes have registers spilling over to shared memory instead of off-chip memory to reduce this devastating effect on performance.

2.3.3 Function Identifiers

A function in CUDA needs to be identified as either a *host function* that runs on the CPU, a *global function* that launches a kernel when called and runs on the GPU, or a *device function* that runs on the GPU and is called by either a global function or another device function.

A function that runs on the CPU is called a host function, just as the CPU is called the *host*. Declarations of such functions may optionally be preceded by the `__host__` keyword, but this is also the default behavior of functions in CUDA so in the event where no such function identifier is given it will default to `__host__`, therefore when creating normal host functions you don't need to specify a function identifier at all.

A function that, when calling it, involves starting up more blocks and threads (by launching a kernel) is referred to as a *global function*. Calling global functions requires a few extra parameters, passed through special syntax, specifying the dimensions of the grid and the dimensions of the block. Typically kernels are launched from a host function, but on newer versions of CUDA, ever since the introduction of dynamic parallelism, kernels can now also be launched from another GPU program.

A function that runs on the GPU and is called from other functions that run on a GPU is referred to as a device function, just as the GPU is called the *device*. Declarations of such functions are preceded by the `__device__` identifier. These functions work similarly to how regular host functions work except that they execute on GPUs, calling them will not launch a new kernel.

In the event you want to create a function that can run on both the CPU and the GPU, you can put both the `__host__` and `__device__` identifiers before the function. Two versions of the function will then be generated by the compiler, one for the host, and one for the device. Such a function can be programmed like any regular function as long as it only utilizes instructions and functionality available for both host and device functions.

2.3.4 Host Programming

Kernel launches

Kernels are launched, typically by a host function, using the following format:

```
kernelFunctionName<<< NUM_BLOCKS, NUM_THREADS >>> ( args );
```

Or if one wants multidimensional block and thread indices one can use the dim3 type:

```
dim3 dimGrid(N_BLOCKS_X, N_BLOCKS_Y, N_BLOCKS_Z);
dim3 dimBlock(N_THREADS_X, N_THREADS_Y, N_THREADS_Z);
kernelFunctionName<<< dimGrid, dimBlocks >>>(args);
```

Memory Management

Memory needs to be allocated on both the host and the device, it must be copied over to the device before the device can use that data, and copied back before the host can access it again. CUDA offers a multitude of functions for allocating and copying memory to the device.

`cudaMalloc(void** devPtr, size_t size)` allocates memory on the device and works in a similar way as the normal `malloc` function in C except it returns an error code instead of a pointer. Therefore, the address of the pointer we wish to assign memory to has to be the first parameter in the function. One can 'free' the memory with a call to `cudaFree(void* devPtr)`.

`cudaMemcpy` is used to transfer memory between host and device. When using `cudaMemcpy`, one has to give as parameters to the function: the destination, source and size of the array, and an indicator of whether you currently want to copy memory from the host to the device or vice versa.

`cudaMemcpy` automatically synchronizes execution between the host and device. If one needs to synchronize execution without copying memory one can call `cudaDeviceSynchronize()` to wait until computation on the device is finished. This is often necessary after a kernel launch if you're using *managed / unified memory* (using `cudaMallocManaged`) where the host and device seemingly share the same global memory, and the moving of memory between host and device is handled by CUDA instead of the programmer.

In the following pseudo-code example we use `cudaMalloc` and `cudaMemcpy` for the allocation and movement of memory for the device:

```
int* a = some_array; // some array of size a_SIZE
int *d_a;
cudaMalloc((void**) &d_a, A_SIZE);
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
kernelFunctionName<<< SIZE / 1024 , 1024 >>>(d_a);
cudaMemcpy(a, d_a, size, cudaMemcpyDeviceToHost)
```

```
cudaFree(d_a);
```

There are many more advanced functions, but I will not describe all of them here.

Streams

Lastly (for the host side), there is also the concept of *streams*, which dictates the order of memory transfers and kernel launches. By default all memory transfers and kernel launches use a single default stream. Sometimes we wish to launch several kernels at the same time; if only one stream is used these kernels will have to wait for each other to complete; by using several streams we can start two streams at the same time, we can also transfer the memory for the use of one stream while the other is executing.

Example of streams:

```
cudaStream_t s1;
cudaStream_t s2;
cudaStreamCreate(&s1);
cudaStreamCreate(&s2);

cudaMemcpyAsync(d_a, a, size, cudaMemcpyHostToDevice, s1);
kernelFunctionName<<< GRID_DIM , BLOCK_DIM >>>(d_a, s1);
cudaMemcpyAsync(a, d_a, size, cudaMemcpyDeviceToHost, s1);

cudaMemcpyAsync(d_b, b, size, cudaMemcpyHostToDevice, s2);
kernelFunctionName<<< GRID_DIM , BLOCK_DIM >>>(d_b, s2);
cudaMemcpyAsync(b, d_b, size, cudaMemcpyDeviceToHost, s2);

cudaStreamDestroy(s1);
cudaStreamDestroy(s2);
```

2.3.5 Device Programming

Indexing

Threads need to figure out their place in the function using their block and thread indices which they will use to index into arrays or as input into if-statements.

Block indices are found in the following way:

```
bx = blockIdx.x;  by = blockIdx.y;  bz = blockIdx.z;
```

Thread indices within blocks are found in a similar way:

```
tx = threadIdx.x;  ty = threadIdx.y;  tz = threadIdx.z;
```

It's also often necessary to find the dimension of the grid or block inside of a CUDA thread. Finding number of blocks in a grid:

```
n_blocks_x = gridDim.x;
n_blocks_y = gridDim.y;
n_blocks_z = gridDim.z;
```

Finding number of threads in a block:

```
block_size_x = blockDim.x;
block_size_y = blockDim.y;
block_size_z = blockDim.z;
```

Absolute thread indices are found like this:

```
my_x = blockIdx.x * gridDim.x + threadIdx.x
my_y = blockIdx.y * gridDim.y + threadIdx.y
my_z = blockIdx.z * gridDim.z + threadIdx.z
```

Shared memory

Sometimes we need to use shared memory, either because it is useful or necessary for the CUDA threads to share data, or because we don't want to run out of register space (forcing the threads to use the slow local memory). To use shared memory one can simply put the `__shared__` identifier in front of the variable or array that you want to be in shared memory. Using shared memory means that all threads in a single block will be able to use the same memory. Example of initializing shared memory on a global or device function:

```
__shared__ int arr[ARR_SIZE];
```

If there is significant reuse of a portion of memory between threads, meaning the same portion of memory is accessed by a large number of threads in the same block, then that memory should probably be loaded into shared memory for faster memory accesses. The threads should share the job of transferring the memory so that one thread does not end up doing all the work. The time required for moving data to shared memory depends to a large extent on the accessing pattern.

One should also be aware that there is a limited amount of shared memory per SM, so using too much shared memory can result in a reduction of the amount of cores that can be run at the same time. If this is the case one may consider using less shared memory.

Synchronizing threads

There might also be a need for synchronizing threads within a block, we can synchronize all threads within a block by putting this statement in our code:

```
__syncthreads();
```

Parallelizing for-loop example

A common feature of parallelizable programs is the use of for-loops. Say we have a double for-loop:

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        out[i * M + j] = in[i * M + j] * 2;
    }
}
```

If we're inside a device function and we have (when adding together all the blocks) one thread for each loop iteration, we can parallelize this double for-loop the following way:

```
// Find thread position
M = gridDim.x * blockDim.x;
int i = blockIdx.y * gridDim.y + threadIdx.y;
int j = blockIdx.x * gridDim.x + threadIdx.x;

// Do the calculation
out[i * M + j] = in[i * M + j] * 2;
```

Note that the same operations happen for multiple threads, this causes the calculation to be done for all correct values for the indices `i` and `j`.

Reduce

It is typical in parallel programs that one wants to combine the results of many threads into a single value using some reduce procedure. CUDA offers resources such as `__shfl_down_sync()` which can be used to implement this efficiently. One might also decide to implement `reduce` from scratch using shared memory and synchronization though for very simple reduction logic this will likely be less effective.

2.4 OpenACC

OpenACC is a higher level interface used for GPGPU programming. It can be used either together with or as a replacement for other GPGPU programming interfaces such as CUDA.

OpenACC is very similar to another programming standard called OpenMP, but it is primarily targeted towards programming hardware accelerators like GPUs, whereas OpenMP is targeted towards multi-core programming on CPUs. They are still very similar and there is much communication between both teams working on these interfaces, and several people who are working on OpenACC have or are also working on OpenMP.

This section is mostly based on the OpenACC programming guide from June 2015 [10], as well as the OpenACC chapter of "Programming Massively Parallel Processors Programming Massively Parallel Processors: A Hands-on Approach" by David Kirk and Wen-mei Hwu [8, pp. 414–441].

2.4.1 Issues with CUDA Addressed by OpenACC

While CUDA is a very effective way to optimize programs for NVIDIA GPUs it has a couple of challenges, it is probably best to just list them outright:

- Writing CUDA code is challenging and requires experienced programmers to do efficiently.
- CUDA programming requires a lot of changes to the code and thus takes a lot of time, even for experienced programmers.
- CUDA is architecture specific, it only works on NVIDIA GPUs, and even then some optimizations that make the code faster on one GPU might make it slower on another GPU.
- CUDA requires a separate source code for the parallel version of the code. Since a computer without a NVIDIA GPU can't run CUDA code it's going to need a separate version of the code that can run on the CPU or another type of GPU.
- CUDA code is hard to maintain. Changes to a program's functionality might manifest in complicated changes for CUDA.
- Older code might become deprecated with time. Someone must manually change the code to take advantage of newer features and remove usage of older features if they are no longer supported or, more likely, just no longer as effective as they used to be.
- Experimentation can require a lot of rewriting. [6]

[6]By experimentation I mean changing aspects of how the program is parallelized by changing the granularity of different loops etc. It could also mean experimenting with changing the function itself, though this is also related to maintainability.

Many parallel programs are similar in nature even if they are different in implementation. It would be better if one could work more abstractly, i.e. work more on parallelizing the actual problems and less on making the program work well on a specific architecture, this is what OpenACC is trying to achieve. OpenACC is made to work on a multitude of high performance computing systems, whether it be multiple CPU cores, or built in acceleration devices like GPGPUs, though it seems to focus mainly on GPGPUs.

OpenACC addresses all the issues detailed above:

- Using OpenACC requires only the addition of compiler directives indicating what parts of the program should be parallelized and how they should be parallelized. Using these directives is often only as challenging as you want it to be, if you want better results you will need to give the compiler more information. One might see speedup with very little effort in comparison to that with CUDA. Using OpenACC should therefore be feasible even for programmers with less experience in performance computing.
- Writing OpenACC code is fast.
- OpenACC is made to be as general as possible, therefore when using OpenACC the code will not be architecture specific.[7] OpenACC offloads work onto the compiler, if you have already specified enough information about how the program should be parallelized all you have to do is just recompile it for another architecture and it will run on that architecture.
- One does not need two versions of the source code to run the program on OpenACC. The OpenACC directives will be ignored if the compiler doesn't know what to do with them or if the architecture doesn't have any parallelization abilities.
- A program that uses OpenACC should be simpler to maintain since OpenACC directives do not significantly alter the source code.
- OpenACC better facilitates maintainability for changes in hardware. Recompilation of the code with a new compiler for a new architecture should automatically take advantage of the new technology available to it.
- Since writing OpenACC requires very little code, experimentation is quick and painless.

[7]OpenACC does have the ability to manually specify architecture specific behavior, but doing so is completely optional. If one does choose to specify architecture specific behavior, this specified behavior will be followed when the program is compiled for the architecture in question, and will be ignored when the program is compiled for other architectures.

2.4.2 Downside with OpenACC

The downside to using OpenACC is of course that you have less control, which limits the potential of optimization, and can lead to the program being slower than what would be possible when programming in CUDA. OpenACC typically works best for more common types of problems, and with some extra directives and clauses it works well for other more complicated problems too. However, if the code you're trying to optimize has a very strange or complicated behavior then the compiler might not be able to find an efficient parallelization of that problem.

If you're an experienced programmer working with a more implementation specific parallelization interface such as CUDA you might very well figure out a better way to parallelize the program for your given programming model than what you would be able to achieve using OpenACC. Given enough time an experienced CUDA programmer will typically be able to write more efficient code in CUDA than in OpenACC, particularly when focusing on a specific GPU architecture, but this will come at the cost of portability, and will also lead to additional time spent programming.

It should be said however that OpenACC compilers are always improving, so a perceived large gap in performance today might be smaller a few years from now. However, since OpenACC is just a higher level interface, it will never surpass in performance what you can potentially achieve using a lower level programming model, but it can provide a very feasible alternative.

2.4.3 Programming OpenACC

When using simple compiler directives, OpenACC will guess how to best optimize the program given the problem at hand, the programmer is welcome to override these decisions using more advanced directives and clauses.

OpenACC's programming model assumes a separation between host and device, each with their own memory, where large amounts of parallelizable computation can get off-loaded to the acceleration device. This does not mean that OpenACC does not work well on unified architectures where the host and device share memory, or where the host and device are in fact physically the same entity. The OpenACC programming model assumes that the host and device are separated since it is easier to adapt from a program assuming a separated model to a program assuming a unified model than the other way around. The programmer is forced to assume that the host and device are separate entities to ease the job of the compiler.

OpenACC treats the data as if it exists exclusively on either the host or the device at any point in time. Whenever computation is off-loaded to the acceleration device, memory has to be copied over, then copied back when the host needs to access the results. The programmer is given additional API functions that can be used to control the movement of data, this is important if one wants to program OpenACC efficiently.

When OpenACC off-loads work onto the acceleration device the work is split into gangs, which are further split into workers, which in turn operate on vectors. Gangs work independently of each other, meaning there is no communication between them except through global memory. Gangs are analogous to blocks in CUDA. Workers are a similar concept to smaller groups of CUDA threads working together in a block, and so they can communicate through shared memory and synchronization. A vector is an instruction performed by several workers at the same time, all operating on the same block of data, and the number of elements the workers work on simultaneously is the vector length. In the case where the vector length is one, a worker is completely analogous to a thread; in the case where the vector length is more than one, a worker is analogous to a group of threads working in unison.

The number of workers multiplied by the vector length is in reality how many threads run in a block. The proportion of the number of workers to the vector length determines how large groups that work together within a gang are. In other words, the proportion of the number of workers to the vector length determines the granularity of the program.

Let's say two nested loops are parallelized, the outer loop is executed in parallel by several workers, and the inner loop is executed in parallel using "vectors" of execution, then prioritizing a high number of workers will lead to a more coarse-grained program, and prioritizing a high vector length will lead to a more fine-grained program (Parallelizing the outer loop is a more coarse-grained approach than parallelizing the inner loop). Exactly what relation between vector size and number of workers is optimal depends on the particularities of the algorithm and the particularities of the underlying architecture; since adjusting such parameters in OpenACC is quick and straight-forward, one can experiment with these parameters until one finds a proportion that works well. One must also keep in mind that if the number of workers and the vector length are both smaller, then the number of threads in a gang is smaller, this might allow the program to run more gangs simultaneously, so there is another trade-off there. This will make more sense in Section 2.4.8, there I will explain how to optimize loops using OpenACC by explicitly delegating work to workers and vectors (specify if a loop should be parallelized by workers or vectors, and how many of them should be used).

In this introductory section on OpenACC I will present how different concepts in OpenACC works in general with very generic pseudocode examples. Later I will use the directives to optimize real programs, this will give a more detailed look into how OpenACC directives work in practice. This will be covered in Section 4.3 and Section 6.7.

2.4.4 Kernels Directive

The `kernels` directive is the simplest way of parallelizing loops. When using this directive you're essentially handing the majority of the work over to the compiler. The compiler will try to optimize the code as best as it can, however it will not be able to make any assumptions about the code that it considers unsafe i.e. it will not make changes it thinks can alter the outer behavior of the program from what was specified by the programmer.

The `kernels` directive is able to do optimizations by itself if it deems it as safe. The `kernels` directive alone will work well on simpler problems, but might perform slowly on more complicated problems in which case it would be a good idea to give further instructions using more directives and clauses.

To get started using the `kernels` directive all one has to do is put the kernel directive before the part of the code you're trying to optimize, then cover that entire area in a set of curly brackets:

```
#pragma acc kernels {
    // parallelizable code i.e. loops
}
```

One can also specify further clauses to help the compiler parallelize the code. Providing hints to the compiler is often necessary to further increase performance, or to convince the overly cautious compiler that parallelization is safe.

2.4.5 Parallel Directive

While the `kernels` directive suggests that loops might be parallelizable, only indicating that the compiler should parallelize them, the `parallel` directive commands it. When using the `parallel` directive it is the programmer's responsibility to ensure that the loops are parallelizable, and if required they should also specify how they are parallelizable. As opposed to the `kernels` directive, where the responsibility of preserving the correctness of the program lies with the compiler, for the `parallel` directive that responsibility instead lies with the programmer.

The `parallel` directive does what it is told to do, and it does not perform optimizations by itself to the same degree as the `kernels` directive. This is useful when testing the program for different levels of optimizations since I will know how much of the program is optimized in each step. The `parallel` directive is therefore the one that I use in this thesis. I will therefore focus mainly on how to optimize a program using the `parallel` directive in this section.

The `parallel` directive specifies a parallel section of the program (this is the part of the program that would need to be rewritten into a global function in CUDA), in this parallel section the program should run in parallel.

The `parallel` directive will indicate the part of the program where the compiler should start a parallel kernel, it is often paired with a loops directive that indicates specific parts of the program that can be performed in parallel (this generally means for-loops). These two directives are often combined into one single line.

```
#pragma acc parallel loop
// for-loop
```

But they can also be separate:

```
#pragma acc parallel {
    #pragma acc loop
    // for-loop

    #pragma acc loop
    // for-loop
}
```

2.4.6 OpenACC Clauses

One must remember to also use the loop directive for the loops when using the `parallel` directive, otherwise the loop will not be parallelized, instead the OpenACC compiler might decide that the entire loop should be run independently by all workers.

We can use the `collapse` clause to indicate to the OpenACC compiler that a specified number of loops should be collapsed into a single loop and parallelized together.

```
#pragma acc loop collapse(NUMBER_OF_NESTED_LOOPS)
// nested for-loops
```

Without extra information the `parallel` directive only works for loops in which the iterations are completely independent of each other. However, there are many common operations in parallel programs, if one of these is used in the code that we are trying to optimize then we can specify this for the compiler. One such operation is the reduction clause, where we need to specify an operation to reduce by and the variable to store the reduced value in.

```
#pragma acc loop reduction(op:var)
// for-loop
```

An example of a typical reduction might be an accumulation of array elements into a single variable:

```
acc = 0;
#pragma acc loop reduction(+:acc)
for (int i = 0; i < N; i++) {
    acc += arr[i];
}
```

Sometimes private variables are also used, for the `parallel` directive we have to specify a variable as private if it is declared outside the loop but should still be considered private by each iteration of the loop:

```
int x;
#pragma acc loop private(x)
// for-loop
```

Specifying private variables in this way is often unnecessary however since any variable declared inside a loop will automatically be private for each iteration of the loop.

One can of course use multiple of these OpenACC clauses for a single loop:

```
int acc = 0;
int x;
#pragma acc loop private(x) reduction(+:acc)
// for-loop
```

2.4.7 Optimize Data Locality

One can optimize the data locality of a parallel program in OpenACC by using data regions. Data regions consist of OpenACC data clauses, and they are used to specify at what point in the program arrays of data should be moved between host and device.

A program optimized with OpenACC directives is supposed to work without data regions included (though sometimes it doesn't), however, the program is likely to run very slowly without using data regions since in that case data movement often ends up being a bottleneck for the program. Data regions are therefore a critical part of optimization with OpenACC, and before data regions are included one is unlikely to see any improvements in the speed of the program using OpenACC.

The data clauses and their explanations are summarized well in the OpenACC programming guide, including these explanations here would be redundant, however I can include an even shorter less detailed summary:

- `copy` - Copy memory between host and device.
- `copyin` - Copy memory from host to device.
- `copyout` - Copy memory from device to host.
- `create` - Reserve memory on the device, but do not copy it between host and device.
- `present` - Indicate that the memory is already present on the device (used when memory has been copied over in a higher level routine / earlier part of the program).

There are also clauses that checks if the memory is already moved over at an earlier point before copying it, these are called `present_or_copy`, `present_or_copyin`, and `present_or_copyout` respectively. The shorthand for these commands are `pcopy`, `pcopyin`, and `pcopyout`, these are the data directives for copying data that I will actually be using. In the OpenACC programming guide [10, p. 30] it is said that the functionality of all data directives will be "*present or*" in the future, and it is recommended to use these directives to ensure correctness with future versions of OpenACC.

Here is an admittedly silly example of how to use four of the data directives:

```
#pragma acc data \
  pcopy(a[N]) \
  pcopyin(b[N]) \
  pcopyout(c[N]) \
  present(d[N])
{
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        c[i] = 4*a[i] + sqrt(b[i]) + 2*d[i];
        a[i] = c[i] * c[i];
    }
}
```

The program doesn't really do anything useful, but is created here for illustrative purposes. In addition to showing how to use data directives it also shows the use of a multi-line pragma, where the pragma continuing on the next line is marked by a backslash.

In this program the 'a' array is used both for input and output, and therefore needs to be copied from the host to the device and then back again.

The 'b' array is only used for input, so it only needs to be copied to the device. After execution is done this memory can simply be thrown away on the device side since the same data already exists on the host side.[8]

The 'c' array is only used for output, therefore it only needs to be copied from the device to the host.

The 'd' array is used as input, however it is assumed that the data is already located on the device. This means that the array was either copied over or created on the device, in some earlier part of the program, likely in a calling function. The array is marked as `present` to make its presence known to the compiler.

One can also copy over parts of the array instead of the whole array by specifying an interval inside the square brackets during a `copy` operation:

```
#pragma acc data pcopy(a[(N/2):N])
{
    ...
}
```

Finally, I will also give an example of what context the `present` clause might be used in:

```
void func1(...) {
    int* arr = (int*) malloc(M * N * sizeof(int));

    #pragma acc data pcopy(arr[M * N])
    {
        for (int i = 0; i < M; i++) {
            func2(N, arr + i*N, ...);
        }
    }
}
```

[8]As far as I know, in the case where this array is changed by the device inside the data region despite memory only being copied in, the array should theoretically be unchanged on the host side, though this might be unspecified. OpenACC is generally used to optimize working serial code however, where the optimization does not alter the behavior of the program in other ways, I would therefore be careful with taking full advantage of this.

```
void func2(int N, int arr, ...) {
    #pragma acc data present(arr[N])
    {
        #pragma acc parallel loop
        // for loop that does something with arr
    }
}
```

Here, instead of moving memory in every call of `func2`, the program instead moves the entire block of memory beforehand, then copies it back once all the GPU computation from calling `func2` is over. This example is only pseudocode, but it is analogous to what is done in the last stage of LSH optimization presented in Section 4.3.3.

2.4.8 Optimize Loops

Earlier I touched upon the trade-offs related to the different proportions for the number of workers vs the vector length, as well as the trade-offs of larger gangs (more threads per gang) vs more gangs able to be run concurrently (fewer threads per gang). The conclusion I made was that these values can be experimented with until one finds a good balance.

This section will be about how to optimize the loops by specifying the number of workers and the number of vectors in a gang, and explicitly delegating work to be performed on the worker or the vector level of granularity.

In order to specify whether the loop should be partitioned among gangs, workers, or vectors, one can use the following loop optimization clauses:

- `gang` - Partition the loop among gangs.
- `worker` - Partition the loop among workers.
- `vector` - Vectorize the loop
- `seq` - Run the loop sequentially (don't partition the loop).

These clauses are used before for loops. The `gang`, `worker`, and `vector` clauses must appear in order, with the `gang` being used for the outer loop, `worker` being used in a middle loop, and `vector` being used for the inner loop. The `seq` clause can be used at any level.

Illustrated examples of how to use these clauses:

```
#pragma acc parallel loop gang
for (int i = 0; i < imax, i++) {
    #pragma acc loop worker
    for (int j = 0; j < jmax; j++) {
        #pragma acc loop vector
        for (int k = 0; k < kmax; k++) {
            #pragma acc loop seq
            for (int l = 0; l < lmax; l++) {
                ...
            }
        }
    }
}
```

```
#pragma acc parallel loop gang
for (int i = 0; i < imax, i++) {
    #pragma acc loop seq
    for (int j = 0; j < jmax; j++) {
        #pragma acc loop worker
        for (int k = 0; k < kmax; k++) {
            #pragma acc loop vector
            for (int l = 0; l < lmax; l++) {
                ...
            }
        }
    }
}
```

In these cases, specifying that one of the loops should be sequential should be optional. Since the `gang`, `worker`, and `vector` clauses are already used there is no way to partition the last remaining loop anyway.

One can also use several of these loop optimization clauses for a single loop:

```
#pragma acc parallel loop gang worker
for (int i = 0; i < imax, i++) {
    #pragma acc loop vector
    for (int j = 0; j < jmax; j++) {
        ...
    }
}
```

Finally, one can also specify how many threads should be used on workers and vector length respectively, one should be aware however that the product of the two values should be below a certain threshold. The product of the number of workers and the length of vectors are the number of threads actually used in a single gang. The number of threads that can be used in a gang is analogously limited as the number of threads that can be used in a block in CUDA. Therefore the number of workers multiplied by the vector length will typically have to be below 1024 for most NVIDIA GPUs at the time of writing; for some newer GPUs the limit might be 2048 instead, it depends on how many threads can run in a single block for that GPU.

Since the underlying hardware is still the same when using a different interface for parallelization, one should still use the same rule of thumb regarding the dimensionality of the gangs/blocks. The total number of threads running in a gang should be a multiple of 32, since this is the size of a warp in CUDA which are groups of threads that work in unison and are more tightly coupled to each other than they are to other threads in the block. In general one tends to use a power of two for both the number of workers and the vector length.

For the `parallel` directive one can use additional clauses to explicitly specify the degree to which each level should be partitioned (gangs vs workers vs vectors).[9] The clauses used are the `num_workers` and the `vector_length`, the program will generate gangs of this dimensionality then run as many gangs as possible given the hardware capabilities.

An example of specifying the number of workers and vectors:

```
#pragma acc parallel loop gang \
    num_workers(64) vector_length(16)
for (int i = 0; i < imax, i++) {
    #pragma acc loop worker
    for (int j = 0; j < jmax; j++) {
        #pragma acc loop vector
        for (int k = 0; k < kmax; k++) {
            ...
        }
    }
}
```

[9]For the kernel directive one can simply add a parameter to the existing clauses in order to specify this which is arguably simpler, but I mostly just cover the `parallel` directive in this introduction.

Chapter 3

Feature Descriptor Matching

3.1 SIFT

As I previously stated in the abstract of this thesis, the SIFT algorithm (developed by David Lowe [9]) finds keypoints in an image, then calculates highly distinctive and invariant feature descriptors from those key points. A feature descriptor is really just a collection of values, typically an array, that are used to describe a local feature in an image, like a key-point. We will explore the sift algorithm in broad strokes in this paper.

The features found using SIFT are entirely invariant to scaling and rotation, and partially invariant to change in illumination and change in 3D viewpoint.

Generally, there are two separate steps to the SIFT algorithm (as well as most other algorithms used for the same purpose):

1. Find keypoints. (keypoint detection)
2. Make distinctive and invariant feature descriptors from those keypoints. (descriptor extraction)

These two steps can each be further divided into more steps, Lowe's paper chooses to divide it into four major stages, some with sub-steps of their own. I myself am going to describe the algorithm in six steps as it is summarized in "Digital Image Processing 4. etd" by Gonzales and Woods [4]. I will however skip a lot of the math and try to describe it verbally in broad strokes instead. Much of the actual information presented will come from the aforementioned book by Gonzales and Woods.

Step 1. Construct Scale Space

The first step of the SIFT algorithm is to construct a scale space of the image. Scale spaces are used by SIFT to find edges as well as to provide invariance to scale changes. Creating a scale space involves creating many different versions of the image, these versions are split across two dimensions: octaves and scales.

Going from an image to an image of one higher octave involves downsampling the image by a factor of 2 then smoothing the image. In SIFT the image is smoothed such that the standard deviation of the image is twice that of the previous image.

Each octave is further divided into many scales. Scales are simply differently smoothed versions of the same image ordered by their smoothness i.e. the standard deviation of the Gaussian filter applied, from least smooth (lowest standard deviation) to most smooth (highest standard deviation).

Note that changing an octave involves both down-sampling and smoothing, whereas changing the scale only involves a smoothing operation. If one is clever with choosing the value of the smoothing factor for the scales one can avoid some extra smoothing operations required when changing octaves and thus save some valuable time.

Step 2. Obtain Initial Keypoints (Scale-Space Extrema Detection)

The next step is to take the Difference-of-Gaussians, that is to take the difference between pairs of differently smoothed images; this will reveal corners, edges, and other interesting parts of the image.

The differently smoothed images used to find the Difference-of-Gaussians is the scale space from Step 1. For each octave we must go through each scale except the first one and take the difference between that image and the one before it. If we have (n) scales per octave we will have (n-1) difference images computed from those scales. The resulting collection of images is a collection of difference-of-Gaussians.

We then have to search for local maxima in the difference-of-Gaussians. This simply means we have to choose pixels from the difference images with more extreme values than all their neighbors, not only neighbors in the x and y axes, but also neighbors in scale (the ones around the same point, but one scale above or below). For one pixel there are 8 neighbors on the same scale, and 9 on both the one below and the one above respectively, totaling in 26 neighbors. If the pixel is either higher than all its neighbors or smaller than all its neighbors, we choose it as an initial keypoint.

The fact that pixels in the difference images have to be compared with neighbors directly above and below them means we have a problem for the first and last difference image, they are lacking 8 neighbors each. In order to combat this we are going to need two extra difference images, for which we need two extra scale images. This combined with the fact that the number of difference images are naturally one less than the number of scales in an octave, means we need three extra scale images if we want to cover a complete octave of difference images. So if we want (s) number of difference images, we are going to need (s+3) number of smoothed images.

Step 3. Improve Keypoint Location Accuracy

After finding the initial keypoints, SIFT uses interpolation to try to find a better keypoint location that, unlike the initial location, does not have to have a discrete x and y location but can instead be somewhere in between the actual pixels.

Step 4. Delete Unsuitable Keypoints

When looking for features we are generally looking for corners and small dots or blobs in the image, small singular points that are characteristic to the object they are captured from. One problem is that a lot of redundant potential points of interest get picked up along edges, cluttering our data. If everything goes as it should this shouldn't happen since a pixel on an edge should be roughly equal in intensity to its neighbors along that edge; however, because there are only a few pixels of roughly equal intensity, this approach is very vulnerable to noise and other irrelevant intensity changes along the edge.

An edge is characterized by a sharp change in intensity in one direction and a low change in intensity in the orthogonal direction. We can use the 2x2 Hessian matrix evaluated at that point in order to estimate the intensity change along the two dimensions. Specifically, if we were to take the eigenvalues of the 2x2 Hessian matrix we should be able to determine whether or not the point is located along an edge by measuring the ratio of the two eigenvalues: if the ratio is close to 1, then it is likely that the point is not located along an edge and we should therefore keep it; however, if the ratio is very high or very close to 0, then it is likely that the point is located on an edge and we should discard it.

Calculating eigenvalues of the Hessian for each initial keypoint would be unnecessarily computationally intensive, and there is a better way: we can find the equivalent of a function of the ratio using the trace and the determinant of the Hessian and check if it is too high, this should save some computation, and serve the same purpose. What we do here is that we specify a ratio limit that we find acceptable, evaluate the aforementioned function using that ratio, then this function value can be used for all comparisons instead of the actual ratio.

Step 5. Compute Keypoint Orientations

This step is very much related to the next step, the order of operation between the two steps is also slightly blurred. The feature descriptors are calculated from the gradient magnitudes and directions in the local area around the keypoint; the problem with this is that we want the feature descriptors to be rotation invariant. In order to achieve rotation invariance we have to assign an orientation to each keypoint, store that information separate from the feature descriptor, then adjust the feature descriptor to be relative to that orientation instead of whatever orientation it gets by default when searching the image.

When the histogram is formed from the local gradients we assign the orientation of the feature descriptor based on the location of the highest peak in the histogram, that is the direction with the highest magnitude in the local gradient. SIFT sometimes chooses to assign multiple orientations if there are several peaks of similar sizes.

Step 6. Compute Keypoint Descriptors

We mentioned before that we calculate feature descriptors from the gradients in the surrounding area around the keypoint. We use pixel differences to estimate the local gradient in a 16x16 region centered on the selected keypoint, and thereby end up with a 16x16 matrix of local gradients (likely actually stored as one matrix of magnitudes and one matrix of angles). The magnitudes of the gradients are weighted by a Gaussian function whose values decrease as it gets further away from the center. The standard deviation of this weighting function is half the size of our region, in this case (when using a 16x16 region) it would have a standard deviation of 8 in both the x and y directions.

Mostly for the ease of comparing features later on, but also for the ease of storing the features, we need to quantize the gradient directions into discrete values, all multiples of 45 degrees apart and covering the entire 360 degree spectrum. This will result in 8 bin histograms where the bins are indexed according to the gradient direction they represent, each holding their respective cumulative gradient magnitude for each 4x4 sub-region of the 16x16 region[1]. In order to fill these histograms, SIFT goes through each of the sixteen 4x4 sub-regions and interpolates the gradient values, splitting the magnitude value of each gradient into the two boxes it falls between, in proportion to which box it is closer to.

After quantizing all the values, accumulating the local gradient values into histograms of 8 bins for every 4x4 sub-region in our 16x16 region around our keypoint, we should then have 8 bins per histogram and 1 histogram per 4x4 sub-region of which there are 16 in total. This gives us a total of $8 \cdot 16 = 128$ bins. All of those 128 bins will be combined into one single data structure and that will be our feature descriptor.

In order to compare these feature descriptors, all you have to do is take the euclidean distance (or some other norm) between two feature descriptors, and given that value you will have to decide whether they are a good match or not. To decide whether two features are essentially the same one might set a tolerance and see if the distance between two feature vectors falls below this tolerance, though typically this is only useful for excluding bad matches. The typical approach for finding a match is to simply find the best fit for the feature descriptor in a data set of other feature descriptors, then perhaps compare the best match distance with a threshold or with the distance to the second best match to determine if the match is significant.

When using SIFT one is generally interested in matching a set of descriptors with another set of descriptors, really what we want to do is to find the nearest neighbor, and we want to do this fast. This brings us to our next section about the nearest neighbor problem and the curse of dimensionality.

[1]The 4x4 sub-regions of the 16x16 region are not overlapping, meaning there are 16 sub-regions in total

Location of feature descriptors in feature space

Though this information does not seem to come from any particular source, it seems that values in feature descriptors all seem to be located in the first quadrant (all values are positive), and on the surface of a hyperbolic sphere around the origin. This will be relevant when discussing how one should most efficiently divide the search space.

3.2 Nearest Neighbor for Feature Matching

The nearest neighbor problem is really a problem of its own, separate from SIFT, but it might also be described as the last stage of SIFT as the ultimate goal of extracting features from an image will be to match them with features extracted from a different image in order to recognize similarities between them.

When discussing the time complexity of the nearest neighbor algorithms I am assuming that one tries to match a set of M points with another set of N points. I will use the variables M and N for that meaning throughout this chapter.

3.2.1 Brute Force

In many ways, efficiently finding the nearest neighbors between a data point and a set of similar data points is a solved problem. There is the brute force method in which you simply iterate through every point in the array, calculate the difference between a point and all the points in the other dataset, and all the while keep track of the minimum difference value and the point corresponding to that minimum difference value. Matching a single point with the dataset will have a time complexity of $O(N)$, matching every single point will have a time complexity of $O(M \cdot N)$

3.2.2 Binary Search

If the points only exist in a single dimension, one can sort the points using a sorting algorithm with time complexity $O(N \log N)$, then use binary search for each point. This has a time complexity of $O(\log N)$ per point. We then have to do that for M points resulting in a time complexity of $O(M \log N)$ for matching the two datasets once one of them is sorted. In conclusion, first sorting one dataset, then matching every data point using binary search only has a time complexity of $O(N \log N + M \log N)$.

Our feature descriptors however do not only have one dimension each, they have 128 dimensions, so using simple binary search is out of the question as they only work for one dimensional data points.

3.2.3 K-d Trees

When dealing with only a handful of dimensions one could use other slightly more advanced methods for efficiently finding the exact nearest neighbor. One such method is using K-d trees. K-d trees were introduced by John Louis Bentley in 1975 [2] and were described as multi-dimensional binary search trees. A K-d tree is a data structure that works sort of like a normal binary tree, but is more complicated in functionality, particularly when searching which requires an extra backtracking phase.

When putting a node into the tree, each level will only compare one of the dimensions of the node with its stored value (a value in the same dimension, taken from a previously added node), generally each level will compare dimension (level [mod] number of dimensions). The idea is that each level splits the feature space in one of the dimensions into two, thus halving the entire feature space, the issue being that a point that is just a little too far away in one dimension might be very close in many other dimensions, and thus we might miss potential matches.

The issue of missing potential matches is solved for K-d trees through the use of a somewhat intricate backtracking phase. During the backtracking phase of the search, the algorithm will, for each level, take both the value of the node that we're splitting our feature space by on this level in the tree, as well as one of the dimensions in the data point we're trying to match with, corresponding to the dimension that this level is splitting. The algorithm will then take the difference between the aforementioned values and check if that difference is smaller than the difference corresponding to our current best fit; if it is smaller, the algorithm will check the other branch of this node as well; if not, the algorithm can carry on up the tree knowing that it has the best match from the branch in which it is located.

For a handful of dimensions K-d trees perform quite well, and are able to maintain the same time complexity as binary search trees with $O(\log N)$ for matching a single point and $O(M \log N)$ for matching an entire dataset.

3.2.4 The Curse of Dimensionality

While K-d trees work excellently for a smaller number of dimensions, the algorithm begins to break down when the number of dimensions increases to larger numbers.

In general, exact nearest neighbor algorithms with smaller time complexities than $O(N)$ to match a single point start to break down with the increase of dimensions in the feature space (number of dimensions for each data point). This rapidly increasing breakdown of performance as dimensions are added is referred to as "the curse of dimensionality".

If we want to match data with higher dimensionality for each data point we essentially have two choices: accept $O(M \cdot N)$ time complexity and solve the problem using brute force; or we can resort to using an approximate algorithm that uses less time but does not guarantee correctness.

These algorithms that trade correctness for efficiency are usually called greedy- or approximate algorithms, an approximate algorithm for finding nearest neighbors is called an Approximate Nearest Neighbor (ANN) algorithm.

3.2.5 Best-Bin-First (BBF)

Best-Bin-First is one approximate nearest neighbor algorithm, and it is presented in Lowe's paper on SIFT [9] as a possible algorithm to choose for matching feature descriptors. It is also explained in more detail in another paper published by Beis and Lowe in 1997 [1].

Basically what Best-Bin-First does is that it cuts the backtracking search for K-d trees short, and thus relinquishes the guarantee of correctness in favor of significantly faster execution time when searching in larger dimensions.

BBF backtracks up the tree and notes down the other branches where the distance from our point to the line that we're splitting by is lower than the distance from our point to the current best fit. It puts these branches into a priority queue sorted by that distance.

BBF then searches the lowest one of these branches, then backtracks again up to the branch it started from, adding new branches to the priority queue if they might contain points closer than the current best fit. It repeats this either for a set number of leaf nodes or until the distance from the current best fit to the closest line dividing up the feature space is below a certain threshold.

In their paper on the BBF algorithm [1] Beis and Lowe showed that BBF worked very well for dimensionality of at least up to 25. They also showed that BBF outperformed the most popular hashing method at the time, though other hashing based algorithms have since entered the scene, such as Locality Sensitive Hashing (LSH), which was introduced by Indyk and Motwani in 1998 [6].

3.3 Locality Sensitive Hashing (LSH)

The presented algorithm for Locality Sensitive Hashing was introduced by Piotr Indyk and Rajeev Motwani in 1998 [6].

Instead of using trees, LSH uses a hashing approach, specifically a variant of hashing that is sensitive to a datapoint's location in the feature space. For each point in a data set, the LSH algorithm finds a value that represents the location of this point in the feature space and groups the points by these values.

In LSH, similar data correspond to similar hash values. Unlike many other hashing functions, where collisions between similar data is to be avoided, in LSH we actually want collisions between similar data. The hash functions are generated using hyperplanes which we use to divide up the feature space. Points falling on the same side of all the hyperplanes have a high probability of being close to each other and are therefore mapped to the same group.

LSH hopes for collisions between similar data, and it tries to bring about collisions between similar data, but it does not guarantee collisions between similar data. Sometimes similar data is mapped to different values because of bad luck; to minimize the chance of this happening LSH repeats the process several times with different hash functions using different groups of hyperplanes.

3.3.1 How the Algorithm will be Presented

In order to intuitively describe the algorithm I will begin by introducing the general behavior of the algorithm by giving a short implementation abstract in Section 3.3.3.

After describing the rough outline of the algorithm I will then move on to the heart of the LSH algorithm, namely the point hashing itself, where I will describe how the algorithm partitions the search space into smaller groups/boxes [2] using hyperplanes.

There will then be a simplified analysis of the time complexity in Section 3.3.7.

In the last two subsections I will then describe in words what the algorithm has to do after finding these hash values. Section 3.3.9 covers how the LSH tables are represented. Section 3.3.10 gives an overview of the final details of LSH.

3.3.2 Terminology

Matching two sets of points is typically done by iterating through the elements of one of the sets, and matching each element with a point in the other set. The set that we are iterating through during the search phase will be referred to as the *query set*, the points in which are referred to as the

[2] I will use the terms "group" and "box" interchangeably; the points that fall into the same box belong to the same group.

query points or the *query vectors*. The query set is the set of points that we are finding a match for.

The other set, the set that is searched each time, will be referred to as the *base set*, the points in which are referred to as the *base points* or the *base vectors*, the base set can sometimes also be referred to as the *search space*. The points themselves are located in a so-called *feature space*.

One can say that the name Locality Sensitive Hashing (LSH) indicates that it only refers to the hashing itself, but I will also use the term for the entire algorithm used for point matching using such hash functions.

3.3.3 Implementation Abstract

A table in the LSH algorithm splits the feature space into many different regions, and maintains a box of points or point references for each separate region in the feature space. LSH goes through every point in the base set and adds it to the correct position in the table, it will check which region it falls into using a locality sensitive hash function and adds it to the corresponding box in a fitting data structure. It will repeat this for every base point.

After putting all the base points into the table it should be clear that the points are roughly grouped by their location in the feature space. This allows LSH to potentially find the nearest neighbor of a query point with reasonable chance by searching through only a subset of the base set.

Once an LSH table data structure is created for the base set, one can try to find matches for a query point by hashing it into the correct box in the data structure according to its location, then comparing it with every base point located in that box.

When matching the query points with points in the same box for only one LSH table the chance of false matches or no matches will be fairly high. A point close to the border might very well match better with a point in the neighboring region than one in its own. In order to reduce the chance of this happening LSH creates several tables, all splitting the feature space in different directions. If a query point is located close to the border of a region in one table, and thus finds a bad match, it will get the chance to find a better match in one of the other tables, where the feature space is divided in a very different way using very different hyperplanes.

3.3.4 Partitioning the Search Space

In Figure 3.1 I have drawn the feature space being split in half by a hyperplane going through the unit circle. The figure is drawn in 2D, but the mathematics related to it that I will present will naturally generalize to higher dimensions.

On the figure I have drawn the feature space as a unit circle. The feature space is cut in half by a hyperplane represented by a vector orthogonal to it, the vector that represents said hyperplane is denoted as **h**.

I have introduced a "positive" and "negative" side of the hyperplane, where the "positive" side contains all points that are further out in the

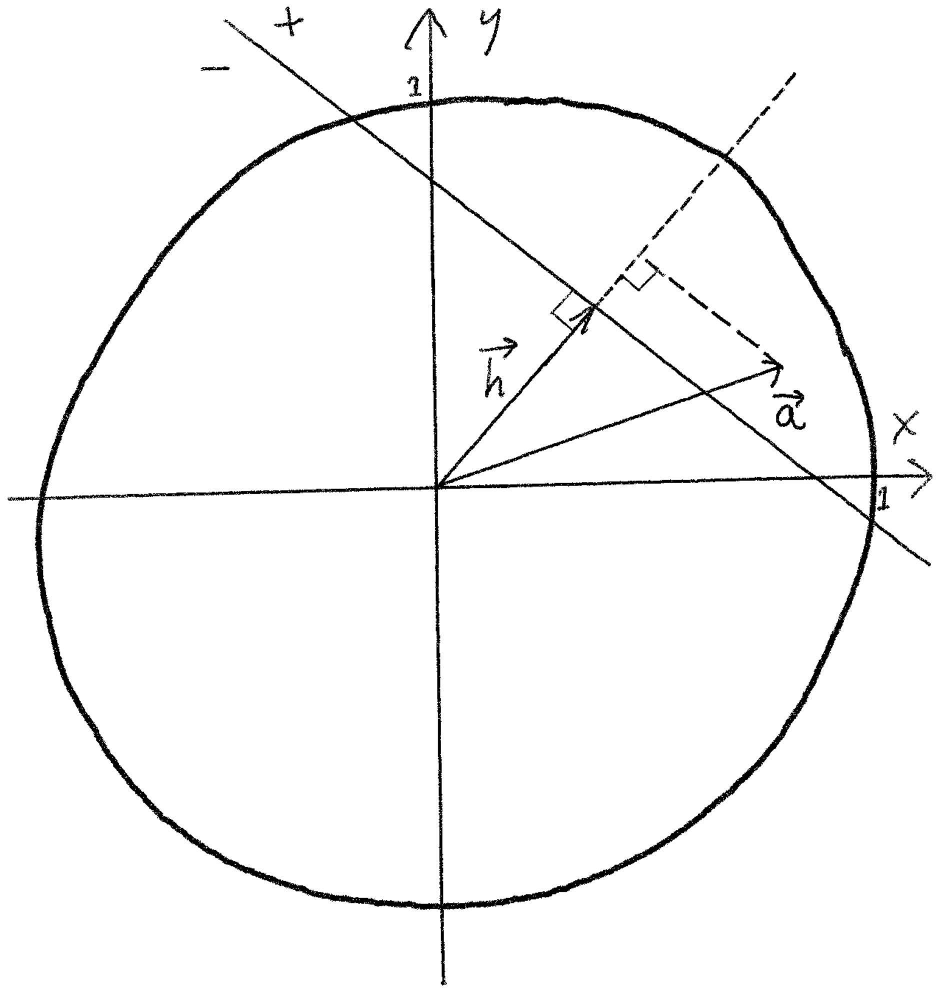


Figure 3.1: Figure showing a hyperplane splitting the feature space in two. **h** is a vector perpendicular to the hyperplane which we use to represent the hyperplane, and **a** is the point/vector we're currently analyzing.
'+' indicates the "positive" side of the hyperplane, i.e. the part of the feature space further out in the direction of **h** than **h** itself.
'-' indicates the "negative" side of the hyperplane.

direction $\mathbf{h}$ than $\mathbf{h}$ itself, and the "negative" side contains all points were this is not the case. The terms "positive" and "negative" are arbitrarily chosen so as to more easily refer to each side of the hyperplane.

In the figure I also drew a second vector which I named $\mathbf{a}$. This vector was drawn on the positive side of the hyperplane, but could just as well have been drawn on the negative side, the purpose of this is to show how one can determine which side of the hyperplane an arbitrary vector in the feature space falls on.

In order to understand how to partition the feature space using hyperplanes one must be familiar with the idea of vector projections. I will not give a complete explanation of projections in this thesis, but I will quickly derive a formula. One technically doesn't need the entire formula for projections, only the scaling factor, but I will introduce it anyway for context.

The euclidean length of the vector $\mathbf{a}$ in the direction of $\mathbf{h}$ can be derived from vector multiplication (dot product):

$$\mathbf{a} \cdot \mathbf{h} = |\mathbf{a}| \cdot |\mathbf{h}| \cdot \cos\theta$$
$$\frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} = |\mathbf{a}| \cdot \cos\theta$$

The symbol θ is here the angle between the two vectors $\mathbf{a}$ and $\mathbf{h}$. The formula $\frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|}$ is equal to $|\mathbf{a}| \cdot \cos\theta$ which simple trigonometry tells us is equal to the euclidean length of $\mathbf{a}$ in the direction of $\mathbf{h}$. This is the scaling factor we need for the projection formula, and also the value we will use to find out which side of the hyperplane $\mathbf{a}$ falls on (this is a signed value, the value will be negative if the projection of $\mathbf{a}$ onto $\mathbf{h}$ goes in the opposite direction of $\mathbf{h}$).

$$(\text{Length of } \mathbf{a} \text{ in the direction of } \mathbf{h}) = \frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} \tag{3.1}$$

By multiplying the length of $\mathbf{a}$ in the direction of $\mathbf{h}$ (i.e. $\frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|}$) with a unit vector going in the direction of $\mathbf{h}$, (i.e. $\frac{\mathbf{h}}{|\mathbf{h}|}$) one arrives at the following formula for the projection of $\mathbf{a}$ onto $\mathbf{h}$:

$$P_{\mathbf{a}\to\mathbf{h}} = \frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} \cdot \frac{\mathbf{h}}{|\mathbf{h}|} \tag{3.2}$$

Now that I have covered the formula for projections (and specifically pointed out that it is divided in two parts, namely a direction and a length/scaling factor) we can now introduce a formula that tells us which side of the hyperplane $\mathbf{a}$ lands on.

Note that a vector $\mathbf{a}$ is on the positive side of the hyperplane represented by $\mathbf{h}$ if the length of $\mathbf{a}$ in the direction of $\mathbf{h}$ is greater than the euclidean length of the vector **h** itself. I.e. **a** is on the positive side if

$$\frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} > |\mathbf{h}| \tag{3.3}$$

otherwise it is on the negative side.

Some further calculations:

$$\frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} > |\mathbf{h}|$$
$$\mathbf{a} \cdot \mathbf{h} > |\mathbf{h}|^2$$

Which in turn gives a simple expression to check if **a** is on the positive side of the hyperplane denoted by **h**:

$$\mathbf{a} \cdot \mathbf{h} > \mathbf{h} \cdot \mathbf{h} \tag{3.4}$$

One can also find the distance from the vector **a** to the hyperplane represented by **h** by subtracting the length of **a** in the direction of **h** from the length of **h** itself:

$$\text{distance from hyperplane} = \left| \frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} - |\mathbf{h}| \right| \tag{3.5}$$

Because of how we chose to represent hyperplanes, equation 3.4 comes with the caveat that it does not work for hyperplanes going through the origin. For a feature space consisting mainly of points of similar lengths (similar distances away from the origin) it might be worthwhile to split the feature space exclusively using hyperplanes going through the origin.

If we want to split the search space using a hyperplane that goes through the origin we can do that by modifying equation 3.4 only slightly, where **a** is on the positive side of the hyperplane if:

$$\mathbf{a} \cdot \mathbf{h} > 0 \tag{3.6}$$

For this equation we're only using the direction of **h** and checking whether **a** is mostly in the same direction or not. It is in a way the same formula as 3.4 with **h** being infinitely close to zero. However, now that we're comparing the dot product with a constant, the length of **h** no longer affects the equation.

Equation 3.6 is a bit simpler than equation 3.4. When representing the hyperplane in this way the equation for finding the difference between vector **a** and the hyperplane is given by:

$$\text{distance from hyperplane} = \left| \frac{\mathbf{a} \cdot \mathbf{h}}{|\mathbf{h}|} \right| \tag{3.7}$$

With equation 3.6, the vector **h** can now be of any length. However, if one wants to more easily find the distance between a vector and a hyperplane then there is an advantage to normalizing **h** by dividing it by its length, effectively setting its euclidean length one. Doing this we can derive an even simpler formula for the distance from the hyperplane:

$$\text{distance from hyperplane when } \mathbf{h} \text{ is normal} = |\mathbf{a} \cdot \mathbf{h}| \tag{3.8}$$

One can also use analogous equations for arbitrary hyperplanes with an arbitrary distance from the origin, including a possible zero distance, by storing the distance from the origin in a separate value, which we can denote r.

That gives the following equation for finding which side point **a** lands on:

$$\mathbf{a} \cdot \mathbf{h} > r \tag{3.9}$$

And the following equation for finding the distance from the hyperplane:

$$\text{distance from hyperplane} = |\mathbf{a} \cdot \mathbf{h} - r| \tag{3.10}$$

In my application of LSH for matching feature descriptors I found it most useful to simply use hyperplanes going through the origin, but implementing a version of LSH that uses hyperplanes with random distances from the origin is quite straight-forward.

The distance from the hyperplane is used if one wants to improve LSH by checking the other side of the hyperplane if the distance is below a certain threshold. This can be very useful for improving the recall of the algorithm without increasing the number of tables or reducing the number of hyperplanes used.

I created Figure 3.2 in an attempt to show how one partitions the search space using hyperplanes that go through the origin where the vector **h** simply gives the direction of the hyperplane. On the figure, I drew **h** as a normal vector; **h** could really be of any length, but in that case one needs to divide by $|\mathbf{h}|$ when measuring a point's distance from the hyperplane as we do in equation 3.7. If one does not care about the distance from the hyperplane, the only thing that matters from the vector multiplication $\mathbf{a} \cdot \mathbf{h}$ is the sign.

Note that all the mathematics presented here holds for general vectors in any dimensions, nothing of what was covered here is specific to two dimensions except for the figures.

3.3.5 Hyperplanes Going Through the Origin: Pros & Cons

Splitting the feature space using only hyperplanes that go through the origin has one major advantage and one major disadvantage.

The advantage is that if the mean of the points is very close to the origin, having a hyperplane go through the origin has a high chance of dividing the search space in two even parts.

The disadvantage is that this will limit the types of hyperplanes that are possible, such that many hyperplanes used in combination to partition the

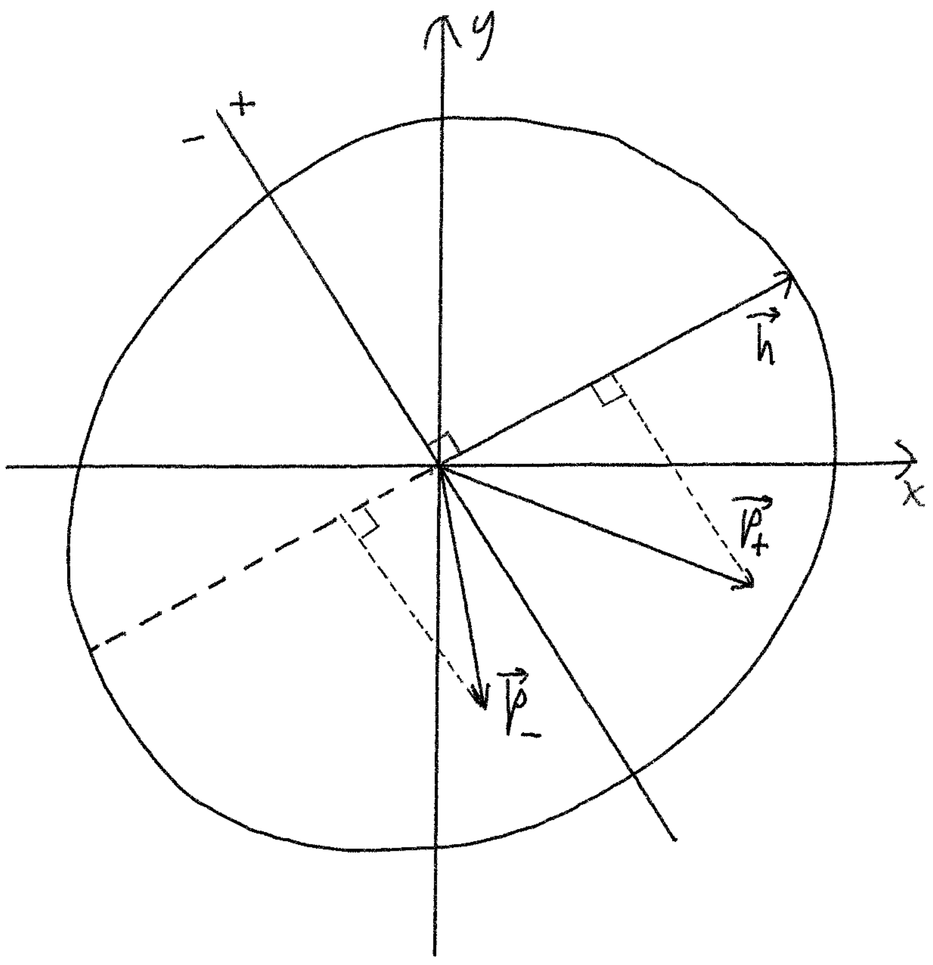


Figure 3.2: Figure showing a hyperplane splitting the feature space in two through the origin. **h** is a vector perpendicular to the hyperplane which we use to represent the direction of the hyperplane.
The hyperplanes distance from the origin is set to zero regardless of the length of **h**.
$\mathbf{p}_+$ and $\mathbf{p}_-$ represents two arbitrary points/vectors where $\mathbf{p}_+$ happens to fall on the positive side of the hyperplane and $\mathbf{p}_-$ happens to fall on the negative side.
'+' indicates the "positive" side of the hyperplane, i.e. the part of the feature space that goes in the direction of **h**.
'-' indicates the "negative" side of the hyperplane, i.e. the part of the feature space that goes in the opposite direction of **h**

.

search space may not partition the search space as efficiently as they would have if they were allowed to be more diverse.

The disadvantage will be particularly severe if the points are located at radically different distances from the origin, as this will lead to points close to the origin being matched with points very far from the origin just as often as they are matched with other points close to the origin. This is due to the fact that if all points go through the origin, the hyperplanes will technically only divide the search space by the angle between themselves and the point in question, if points are equally far from the origin, this doesn't matter, but if they vary in length then this is not optimal.

For the application of matching SIFT feature descriptors, the disadvantage is negated by the fact that the feature vectors are all of equal length, i.e. they have an equal distance from the origin. They will all be located on the surface of a sphere around the origin in the first quadrant with only positive elements. Meanwhile, the number of dimensions in a feature descriptor is very high compared to the number of hyperplanes used for a single table, there should therefore be enough dimensions for the hyperplanes to be very diverse even with some restrictions on their directions.

The locations of the feature descriptors also seem to negate the advantage that I listed. Because the feature descriptors only contain positive elements and are located away from the origin, the mean of the feature descriptors can obviously not be zero. The best way to fix this is to create hyperplanes that are specifically designed to cut the first quadrant of the feature space such that the feature vectors can keep their position in the feature space (the feature vectors being equally far away from the origin is an advantage that should be kept if possible). One can create such hyperplanes by restricting them such that their elements sum to zero, ensuring that all hyperplanes will always split the first quadrant of the feature space in two equal halves. I will discuss this further in Section 3.4.2. When such hyperplanes are used, having the hyperplanes going through the origin can be beneficial also for matching feature descriptors.

For my implementation of LSH I will therefore use hyperplanes that only go through the origin, as this is beneficial when matching feature descriptors.

3.3.6 Calculating Hash Codes

Now that I have explained how to calculate what side of a hyperplane a point falls on, it is time to explain how to use this information to construct hash codes which we can use to group the points in a dataset together.

In order to divide the feature space we use several hyperplanes, each splitting the space in two in separate directions, then only points that are on the same side of every one of those hyperplanes are given the same hash value and are thus grouped together. The hyperplanes are combined into a list, where $\mathbf{h}_i$ is hyperplane number i in the list, starting from 0.

The number of hyperplanes can generally be of any number, but they are often set to $\log_2(N)$ where N is the number of base points. The reason for this is that when setting the number of hyperplanes to $\log_2(N)$ one can show that the time complexity of the search algorithm for finding the best match for a single query point will be logarithmic.

One does not have to use exactly $\log_2(N)$ hyperplanes when running an LSH algorithm, but if the number of hyperplanes is set to $\log_2(N)$ then the algorithm should theoretically have a logarithmic time complexity assuming the hyperplanes are effective at partitioning the search space.

Constructing hash codes using the aforementioned hyperplanes is pretty straight forward. For a single point **a** we can calculate which side of each hyperplane the point falls on using vector multiplication as I explained in the previous section. For every hyperplane $\mathbf{h}_i$, if **a** is on the "positive" side of $\mathbf{h}_i$ we represent this as a 1, on the other hand if **a** is on the "negative" side of $\mathbf{h}_i$ we represent that event as a 0.

We can then combine those ones and zeros into a single integer by using them as bits, starting from the least significant bits. We can generate this hash code by doing the following: first we initialize the hash code to zero, then for every hyperplane $\mathbf{h}_i$, if **a** lands on the "positive" side of $\mathbf{h}_i$, the i'th bit is set to 1; if **a** lands on the "negative" side of $\mathbf{h_i}$ the i'th bit remains 0.

```
hashcode = 0;
for all i
{
    if (dot_product(a,h[i]) > 0)
    {
        hashcode = hashcode | (1 << i);
    }
}
```

By calculating the hash-code we have effectively assigned **a** to some group that corresponds roughly to where **a** is located in the feature space.

These hash-codes can later be used to group the points together so they can be matched much faster with points from other datasets. The points are grouped into tables, and the hash codes are used to find the correct box in a table to store the point in for a given base point, or to search in for a given query point.

One nice property of LSH is that the closer a query point is to its best match in the base set, the higher the probability is for those two points to be matched. Therefore, if two points are more likely to actually be a true match then they are also more likely to be matched together by LSH. It also means that if a query point is not very similar to any of the base points, then the probability of finding the best possible match among them decreases, this is not so important however, since in this case the best match in the base set was not a very good match anyway.

3.3.7 Time Complexity: Simplified

In the paper on LSH [5], different aspects of the algorithm such as the time complexity and the error probability are covered in great detail. In this thesis however, I will only present a smaller, perhaps simplified, example heavily based on information that I found in an online lecture by Victor Lavrenko.

To find the computational cost of the algorithm, let us say there are N points. Each point is in D dimensions, and the hashing algorithm uses H number of hyperplanes. Finally the number of tables is denoted by T.

In order to calculate a hash code of a single point, we need to, for each hyperplane, calculate which side of the hyperplane the point falls on, which is done using vector multiplications. A locality sensitive hash code is created by H vector multiplications between the point and the hyperplanes. Each vector multiplication requires D multiplications and $D - 1$ additions, or just D multiply and accumulate operations. We therefore say that calculating a hash code for a single point requires something on the order of $D \cdot H$ primitive operations.

Since there are N points divided between 2^H boxes, if one assumes the points are evenly split between the boxes[3] there will be $(N/2^H)$ points in each box.

The number of base points that a query point has to be compared with when using a single LSH table will then be something on the order of $(N/2^H)$. Comparing a point requires D subtractions and multiplications, as well as $D - 1$ additions, but let's pretend for simplicity that it is D primitive operations. The number of operations to match a single point once it is grouped into the correct box will then be $D \cdot (N/2^H)$ primitive operations, this is the time used for comparisons.

Both the hashing and the comparisons for a point will have to be repeated for each table T. The time complexity for finding a match for a point will therefore be multiplied by T.

Time complexity for finding a match for a single query point:

$$T \cdot (D \cdot H + D \cdot (N/2^H)) = TDH + TDN/2^H$$

The first component of the time complexity is only a multiplication of the number of tables, the number of dimensions, and the number of hyperplanes; all those factors can be controlled so that the total cost is fairly low. The last component of the time complexity depends on N and therefore grows linearly, but the expression is divided by 2^H. To make the time complexity logarithmic one sets the number of hyperplanes to be equal to the logarithm of N such that the N factor gets canceled out.

$$\text{Set number of hyperplanes, } H := \log_2(N)$$

One can then show that the time complexity of the algorithm becomes logarithmic:

[3]The points being close to evenly divided between the boxes is highly unlikely, which is actually a problem when using LSH, and a reason why creating good hyperplanes is such a big undertaking in and of itself. It is assumed here however, since it makes it possible to reason about the time complexity of the algorithm.

$$\begin{aligned}\text{Time complexity} &= TDH + TDN/2^{H} \\ &= TD\log_2 N + TDN/2^{\log_2 N} \\ &= TD\log_2 N + TD \\ &= TD(\log_2 N + 1) \\ &= O(\log_2 N)\end{aligned}$$

The number of dimensions and the number of tables were here treated as constants.

One big caveat of increasing the number of hyperplanes to reduce the time complexity, is that increasing the number of hyperplanes does not only reduce the time taken by the algorithm, it also reduces the probability of finding the best match for a given query point. Each hyperplane added has a certain probability of coming between a query point and its closest match, thus dividing them into two different groups. That being said, the algorithm is still very effective, and the closer a query point is to its best match, the more effective it is, since the probability of a hyperplane dividing the two points into different groups becomes lower the closer the query point is to its best match in the base set.

In addition to creating mathematical models for the time complexity, one can also create mathematical models for the probability of error given a certain number of tables and assumptions about the input points. I will stop the algorithm analysis here however, with this simplified view of the time complexity, as an in-depth analysis quickly becomes very complicated, and therefore falls out of scope for this thesis.

3.3.8 Number of tables

One thing that has to be decided when running LSH is the number of LSH tables to use. Choosing the number of tables involves a very important trade-off, using more tables leads to a higher probability of finding a point's nearest neighbor, but using more tables also results in more computation since the search has to be repeated for each table.

The exact number of tables to choose thus depends on balancing the importance of speed vs. accuracy; however, you cannot simply choose the speed and accuracy that you require. The speed of a single search depends not only on the number of LSH tables, but also on how many base points fall into the same tables as the query point, this in turn depends on how well the feature space is partitioned and how our point datasets are distributed around this search space. Meanwhile, the accuracy of the algorithm cannot be predetermined either since it depends both on the distance of our point from its closest match, and also on just random chance. If one makes assumptions about the data, one might use advanced mathematical models to make some presumptions about the performance of the algorithm based on probability, but these presumptions might not be valid if the data deviates sufficiently from what was assumed.

Since the number of tables is only one parameter however, one can just experiment with a few different numbers until one finds a number of tables that provides the right combination of speed and accuracy for some typical input.

3.3.9 How To Represent LSH Tables

Direct or Indirect Pointers

One important design decision that has to be made when implementing the LSH table data structure is whether to use indirect pointers to the points in an original list, or copy the points directly into each LSH table. There are advantages and disadvantages with both approaches.

Using indirect pointers has the following advantages:

- There is much less memory used. Because we use a large number of LSH tables for a single dataset, and the points or point references have to be repeated for every table, when using indirect pointers we will only need one memory address for each point instead of 128 floating point values as would be the case if points were stored directly in each LSH table.

- Since only the indirect addresses must be stored separately for each table, and the base points themselves remain constant in memory, much of the same data will be reused when matching a point with each table. This memory reuse will likely be good for caching, which improves the performance of the algorithm. (Though an extra level of indirection itself is bad for caching which might reduce the performance for the same reason.)

- Another advantage is that creating the LSH tables will be slightly faster. This is also because the LSH tables are using less memory. It is faster to assign an indirect address than it is to move an entire feature descriptor with 128 values. (We also won't need to actually look up the addresses while creating the tables, this will save some time when creating the tables, however it may or may not lead to more time taken during matching.)

- Finally, and perhaps most importantly, using indirect pointers for the feature descriptors allows for an optimization of the program by finding potential matches for each query point before the actual matching, which enables us to ignore duplicate point comparisons across tables.

  I will discuss this optimization in detail later in this thesis, but it makes it possible for the algorithm to avoid comparing the same points more than once if a query point matches with the same base point in several tables, something that is particularly important when using many tables since points that are grouped together in one table are likely to be grouped together in other tables as well.

Finding possible matches before actually comparing the points also makes the program easier to parallelize, as functionality can more easily be split into separate, much simpler functions.

The potential advantage of using direct pointers is mainly during matching, given that we are comparing it with the "naive" matching routine where we do not check for potential matches beforehand. We don't want the program to go through levels of indirections unnecessarily. Accessing memory takes a lot of time, particularly if that memory is not cached. Chasing pointers is very likely to decrease performance, particularly if it occurs within an inner loop.

While there is an argument to be made for storing the points directly in the LSH table, I believe this argument is less convincing than the arguments listed for using indirect pointers instead. I therefore opted for using indirect pointers for the advantages of using indirect pointers that I listed above.

LSH Table Organization

Each LSH table consists of a collection of the following arrays: an array of group indices, an array of point references sorted by groups, the points themselves, and optionally, an array of group sizes (though this information is implicit from the indices and the number of groups).

First we need an array containing the points themselves, or in our case where we use indirect pointers, this array instead consists of pointers to the points in our matching dataset. The point references [4] in this array are not sorted by their original order, instead they are sorted by their groups with points belonging to the same group grouped together. Because the point references are ordered in terms of their groups I decided to call this array `groupArray`.

We also need an array storing the indices used to index each group in `groupArray` given its corresponding hash code, for this I created an array called `groupIndexMap`. `groupIndexMap` is indexed by LSH hash codes of points and returns an index for the correct group in `groupArray`, or rather the first element of that group. In order to search the group we also need to know how many points are in that group.

In order to keep track of the number of elements for each group we can use an array storing the sizes of each group, I decided to call that array `groupSizeMap`. `groupSizeMap` is useful for creating `groupIndexMap`, but once `groupIndexMap` is created one technically doesn't need `groupSizeMap` anymore since one can instead subtract the current index from the next index (or max index for the last element) and get the size that way. One can make this work even for the edge case by adding an element to the end of `groupIndexMap` which contains the size of `groupArray`.

In my implementation of the normal LSH algorithm I kept the `groupSizeMap` array, mostly because of path dependency of my codebase since the program was already implemented in that way and changing it would require extra effort with little change in performance. I will

[4]"references" in the normal meaning of the word, not C++ terminology.

also present another implementation of LSH that only uses one table, but achieves an acceptable accuracy by checking neighboring boxes; that version discards `groupSizeMap` after creating the LSH tables.

I created Figure 3.3 in order to visualize how one can represent LSH tables using

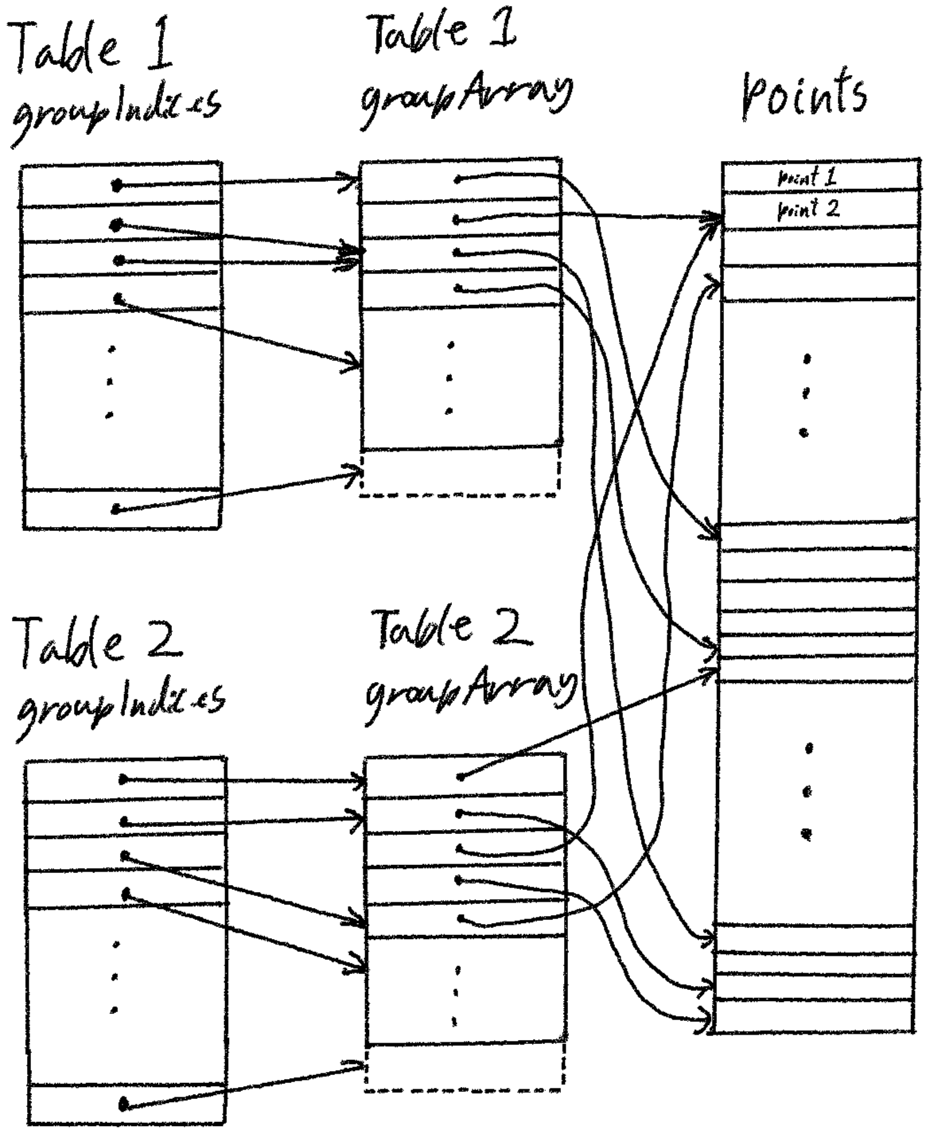


Figure 3.3: This figures visualizes the organization of a set of two LSH tables. Note that the last element of `groupIndices` points to the address after the last element of the corresponding group array, this way the size of each group can easily be found by subtracting the current index from the next index.

3.3.10 LSH Algorithm Overview

The LSH algorithm can be split into two main parts: constructing LSH tables, and looking up points in the table. While not strictly necessary, it is convenient to keep these two functionalities separate for the user so they can reuse the LSH tables if they want to search the same base set using another set of query points later on.

Both parts of the algorithm, table construction and matching, will have to calculate hash values in order to find a point's group mapping.

When constructing the LSH tables one of course needs to use a set of base points, likewise, it can be a good idea to group together the query points into their own dataset before matching so that the task of matching them becomes more parallelizable.

Constructing LSH Tables

In order to construct the hash table one first needs to calculate the hash value for every point in the base set and store the hash values in an array which I decided to call `indexGroupMap`.

I created a function that I called `organize_points_into_groups`. This function takes `indexGroupMap` for a single set of hyperplanes as input and uses it to create the arrays necessary to represent a single LSH table. `organize_points_into_groups` is called by another function named `construct_lsh_tables` which first calculates the hash values for every base point using the hyperplanes in the LSH table, then calls `organize_points_into_groups` for each LSH table to organize the points into groups and thus create LSH tables.

Among other things, `organize_points_into_groups` sorts the points by the group they are mapped into, usually such a sorting operation has a time complexity of at least $O(NlogN)$, but due to the upper and lower limits for the hash values, `organize_points_into_groups` can do it in linear time, this would not be possible if the interval of possible hash values was impractically large.

`organize_points_into_groups` is a fairly simple function and it is split into three parts.

First it constructs an array to store the size of each group (named `groupSizeMap`), this is done by going through the `indexGroupMap` array, checking which group each element falls into, then it increments the size counter for that group.

After creating `groupSizeMap`, the function uses that data to create `groupIndexMap`. `groupIndexMap` is simply just an exclusive scan of `groupSizeMap` and is therefore very easy to create. The function also creates a copy of `groupIndexMap` called `groupTailsMap` which allows us to more easily add elements to the end of the group when creating the array with pointers to the actual points which I called `groupArray`.

`groupArray` is created by simply going through each point one by one, checking which group it falls into, then pushing it onto that group using `groupIndexMapTails` to keep track of the last element of each group.

Once `groupArray` is created, `groupIndexMapTails` no longer contains any useful information and can be discarded. As mentioned earlier, contents of `groupSizeMap` is also implicit from `groupIndexMap` and the number of points, `groupSizeMap` can therefore also be discarded if one chooses to do so.

Matching Points

In order to more easily parallelize the algorithm, while also avoiding duplicate point comparisons when a query point matches with the same points over multiple tables, I decided to split the matching process into two parts.

The first phase of point matching is to find potential matches, i.e. go through each table for every query point and add the the base point indices that are found inside the LSH table to a set of potential matches where duplicates are excluded. These sets should be individual for every query point.

In order to check whether or not a point is already inside the potential matches set I use an integer array where an element is equal to the index of the current query point if the point at that index in the base set is already in the set of potential matches. I decided to use an integer array instead of a bitmap; the integer array uses a bit more memory but also comes with the advantage that the algorithm won't have to reinitialize the map for every single query point, this is important for the performance of the algorithm. I decided to call the function that finds the potential matches `find_potential_matches`.

All of this is done serially since parallelizing the function using GPUs is impractical. This can be parallelized, though it would probably need to be done in a more coarse-grained fashion. More details as to how this function is implemented is found in Section 4.3.5.

The advantages of finding potential matches separately from doing the point comparisons are two-fold:

1. The matching phase can avoid duplicate comparisons between the same two points which would otherwise occur frequently.
2. It simplifies the job of actual point matching, making that part of the algorithm easier to program for GPUs.

Inversely, the disadvantages of skipping this part of the program by combining this functionality with the functionality of the "match_points" function:

1. When many tables are used, the chances for duplicates point comparisons in the match phase will be very high, and there will be very many of them, this will significantly increase the time taken during matching.

2. It becomes much harder to divide the work-load fairly among the threads during parallel execution in the matching phase since some boxes in the LSH table will be much more populated than others which can lead to an unpredictable amount of work for each thread depending on how the program is parallelized.

Once `find_potential_matches` is finished running we should have many sub-arrays stored contiguously in a large array. The array is called `potentialMatches` and it is indexed using another array called `potentialMatchesIndices`. Each sub-array contains indices of base points in the base set, these are the potential matches for each query point stored.

Finally, after we know the potential matches for each query point we can call a function to compare each point with its potential matches, I decided to call this function `match_points`.

`match_points` works in a very straightforward way. First, it iterates through every query point, for every query point it will find its potential matches by looking up indices in `groupArray`, then go and compare each of the potential matches one by one. The function keeps track of the best and second best matches for each point as well as its distances which will be stored as output in an output array.

3.4 Good Hyperplanes

In this section I will go discuss various problems that can occur when trying to split the search space using hyperplanes. I will also discuss some loose ideas as to how to solve these problems. The last two subsections are not as important as the rest of the thesis, but they are included since they provide some relevant discussion.

In addition to the subjects presented in this section there is also an algorithm presented in Section 6.6 about creating a good set of orthogonal hyperplanes. That algorithm is presented in Chapter 6 and not here since orthogonal hyperplanes pertains particularly to the single table LSH algorithm presented in that chapter.

3.4.1 Problem of Clustered Input

A challenge one can run into when splitting the feature space might arise if the points in the matching set are clustered together; this is particularly common for real applications with images of many similar objects. Points being clustered together might lead to very many points being assigned to the same group.

The worst case scenario of this is if all points in the matching set falls into the same group, in this case LSH will do the same thing as a normal linear search just less efficiently. All the points falling into the same box is of course highly unlikely, but the fact that points clustered together might make some boxes overpopulated is an indication of a bigger issue.

For random hyperplanes, the more clustered the points are, the more time the hashing algorithm will use as there are more potential matches that each point will have to be compared with.

If all the input data is clustered around a single point, one can easily solve this problem by simply subtracting the mean of the base set (performing mean normalization) before constructing the tables, then storing that mean so it can also be subtracted from the query points before trying to match them with the base points in the LSH table. Once the mean is subtracted a hyperplane is more likely to actually split up the cluster.

Mean normalization should solve the problem if the points are all clustered around the same area, if however, there are two clusters on opposite sides of the origin, mean normalization won't help as neither of the two clusters will move.

When using SIFT feature descriptors however, the points are not evenly divided around the unit surface, but are rather located on the surface of a hypersphere in the first quadrant of feature space (containing only positive elements). If the hyperplanes are properly adapted for it, having the points be positioned like this can actually be a big advantage for LSH, particularly when using hyperplanes that all go through the origin, as there will then be no points located close to the origin, and thereby also close to all the hyperplanes. For SIFT features it is therefore likely a bad idea to perform mean normalization if it's part of a greater procedure that might produce other benefits in return.

Having only a few larger clusters can be bad for performance, but on the other hand, if there are very many clusters, and each of them are relatively small, and the clusters themselves are evenly distributed throughout the feature space, then the algorithm will do exactly as it should, it will partition the feature space, and points within the same clusters will likely end up in the same group. In this case the algorithm is working as it should. In fact, dividing the search space into smaller clusters is exactly the type of functionality we want the algorithm to achieve, it is only a major problem when those clusters are too large.

In other words, all the elements of a cluster falling into the same group is only a problem if the sizes of those clusters are very large compared to the number of boxes we're hashing points into.

When we partition the search space using hyperplanes, the hope is that the search space is spread as evenly as possible between the boxes. The other goal is that the points are not located along the border, i.e. that their distance is as far away from the hyperplanes themselves as possible.

In the case where there are a few very large clusters, the most important task will still be to separate the clusters from each other. After that it will be important to separate the clusters themselves to avoid having to linearly search through the entire cluster.

Finally, when it comes to dividing the search space efficiently, the hyperplanes chosen may have a large impact on the result, there might therefore be some advantage in trying to choose more fitting hyperplanes that more effectively partitions the search space so that the points are spread more evenly between the boxes.

3.4.2 Problem of Finding Good Hyperplanes

Finding a new set of hyperplanes that partitions the search space well for every new dataset would be computationally intensive, and it would only really be worth it if the LSH tables would be reused a large number of times. If one wants to avoid finding new hyperplanes for every new set of base points one can instead try to find hyperplanes that partitions the search space really well for some typical input and then applying those hyperplanes to new data.

There are many ways to try to most effectively partition the search space for a typical type of input. Generally, what one would be looking for are hyperplanes that split the dataset into groups such that each group has a similar amount of elements, while doing this it is also good if the LSH tables partition the search space in ways as different from each other as possible, meaning the hyperplanes used by one LSH table should be very different from the hyperplanes used by all the other LSH tables. That would help the LSH tables to better compensate for each others' faults such that nearby points that are not grouped together in one table still have a high chance of being grouped together in another table.

There are many ways of finding hyperplanes that are good at partitioning the search space for some example dataset. If one decides to generate hyperplanes specialized for every individual dataset during run-time, then one needs to be very conscious of the time used to generate those hyperplanes. However, as long as the hyperplanes are precomputed and not decided during run-time, a program can generate these hyperplanes using almost as much time as it likes (within reason of course).

As mentioned several times in this thesis, the SIFT feature descriptors are located on the surface of a hyperbolic sphere in the first quadrant of the feature space. This means that every SIFT feature descriptor has approximately the same distance from the origin, and all the vector elements of the feature descriptors are positive numbers. Since all the vector elements are positive, it is clear that a hyperplane with, for example, an overabundance of positive elements will yield results where the feature descriptors are all much more likely to be mapped to the positive side of the hyperplane than the negative side. Likewise, for hyperplanes with an overabundance of negative elements, the feature descriptors will be much more likely to be mapped to the negative side of the hyperplane than the positive side.

For random vectors that contain positive or negative elements with roughly equal probability, completely random hyperplanes will split the search space efficiently. When we want to split feature descriptors where all the elements are positive however, one has to think a little differently, a hyperplane going through the origin represented by a vector with all positive elements will simply put all the points into the same group, and therefore not split the search space at all, the hyperplane does not split the first (all positive) quadrant. If one moves the hyperplane away from the origin, it will start to split the points into two groups, but since the points are located on the surface of the all positive section of a hyperbolic sphere, the hyperplane will not divide the search space in a way that is very locality-sensitive when it comes to points on the negative side of the hyperplane.

In order to better divide the search space for SIFT feature descriptors, where all elements are positive, it is a better idea to guarantee that the hyperplanes will cut through the first (all positive) quadrant, at least to some substantial degree. One way to do this is to make sure that the vector representing the hyperplane will have roughly equal positive and negative weights, in other words, that its elements will sum to a value that is not too far from zero, exactly how far away from zero depends on a trade-off between how well each hyperplane will be able to split the all-positive part of the feature space, vs. the degree of freedom to which the hyperplanes are allowed to vary from each other. In short, the trade-off is between individual effectiveness and diversity of hyperplanes.

What I decided to do was to, for each hyperplane, make its elements sum exactly to zero, as that gives each hyperplane the best chance of splitting a search space with all positive elements into equal parts. This means deprioritizing some possibility of extra hyperplane variability in favor of individual effectiveness. I was comfortable making this trade-off because SIFT features have 128 dimensions, such high degrees of freedom allows for plenty of variability between hyperplanes even with certain restrictions.

The best way to create a random hyperplane that sums to zero is to create two random hyperplanes, calculate the sums of both, calculate the multiplication factor that would make the sum of the second hyperplane equal to that of the first hyperplane, then subtract that factor times the second hyperplane from the first hyperplane, then finally discard the second hyperplane. The final hyperplane is then made to be of length 1. There are a few very rare cases where this doesn't work, such as if the final hyperplane only contains 0 elements, or the sum of the second hyperplane starts off as 0, but these cases are extremely unlikely.

Illustration of how to create a random hyperplane with elements that sum to 0, h_3:

$$\begin{aligned} h_1 &:= \text{random hyperplane} \\ h_2 &:= \text{random hyperplane} \\ h_3 &:= h_1 - \frac{\sum h_1}{\sum h_2} \cdot h_2 \end{aligned}$$

This subsection has mostly been about how to most effectively divide the all-positive feature space into equally sized parts using random hyperplanes. In the next subsection there will be some more discussion on more effectively dividing the search space of the feature descriptors which are generally not at all evenly divided around the feature space.

3.4.3 PCA for LSH

Principal Component Analysis (PCA) is an algorithm used for dimensionality reduction, very commonly used in statistics and data science. I will not describe in detail how PCA works in this thesis, but I will describe the goals of PCA, and why it might be useful for the point matching problem in general, and more specifically why it might be useful in combination with LSH.

As covered in the chapter on SIFT, individual dimensions of the SIFT feature descriptors all directly represent some specific aspect of the image relating to the local gradient of a perceived point of interest. This representation provides very distinct features, and the features themselves represent something concrete that can be found in the image. There might however, be a great deal of redundancy in this representation, when data for example is clustered around certain regions or certain lines there is reason to believe there might be some redundancy in the data.

One can imagine that, instead of using the standard basis vectors of these feature descriptors, one can instead represent those same feature descriptors using a combination of a smaller number of basis vectors by performing a change of basis into a lower dimensionality space, thereby approximating the feature descriptors and reducing the dimensionality for faster vector comparisons and multiplications.

What PCA tries to do is essentially finding the directions that contain the most information in the existing high dimensional space, then using vectors along these directions as basis vectors instead of the standard basis.

The principal components of a dataset is found by first subtracting the point mean from all the points in the dataset (mean normalization), then finding the direction of maximum variance and creating a unit vector in that direction; that vector is the first principal component. After that it will iteratively first subtract from each point in the dataset their projection onto the last found principal component, then find the next direction of maximum variance, create a unit vector in that direction, and make that the next principal component.

After using PCA to find the principal components one then has a set of vectors that are all orthogonal to each other, and each of them points in the direction of maximum variance, under the constraint that each of these vectors must be orthogonal to all the other vectors before it in the list.

Each principal component also comes with a scalar indicating the amount of variance in their direction, this scalar indicates the importance of the principal component in question and will decrease in value the further down the list of principal components one goes.

One has to be aware however that the first step of PCA is mean normalization, as the theory behind PCA doesn't work without mean normalization. Therefore if one applies PCA on feature descriptors, the points will no longer be located on the surface of a subset of a hypersphere, but will instead be more evenly spread out throughout the space around the origin, and elements of the feature descriptor vectors may have either positive or negative values.

Calculating PCA for each new set of base points that one receives as input is probably way too computationally intensive to be worthwhile. Since different large sets of feature descriptors often seem to resemble each other in their placements in the feature space however, it might be worthwhile to calculate the principal components of some large typical set of input, then use those principal components for dimensionality reduction for new points before applying LSH or other matching algorithms.

One also has to remember that for every change one makes to the base points, one must make equivalent changes for the query points such that they can be matched.

For application specific circumstances, where similar images with similar features need to be matched over and over again, reducing the dimensionality of the data to work better for these specific features can be very useful in order to improve speed.

There are two different ways, I believe, that one can use PCA for improving LSH. One is for dimensionality reduction, and the other is using the first few principal components directly as hyperplanes (for both methods one has to perform mean normalization on the training, base, and query sets).

Dimensionality reduction is probably the best way to use PCA before applying LSH, particularly if one wants to use several tables. If one uses a single table one might simply use the principal components as hyperplanes, which allows for bypassing an entire dimensionality reduction phase of the program prior to calculating hash values, a major disadvantage being that some of the hyperplanes in the list being more important than others while still only representing a single bit in the hash code.

Dimensionality reduction for LSH

One way to reduce the amount of work of point matching might be to reduce the dimensionality of the data before applying a point matching algorithm. The presence of fewer dimensions alone will make point comparisons faster, for LSH it will also make the hashing faster.

One can also argue that reducing dimensionality beforehand might make the hashing algorithm more efficient since the points are then likely to be more evenly spread out in the remaining space, i.e. if there is a dimension along which all the points are clustered together and the points don't change much, this dimension will likely be considered as redundant by PCA and discarded. Completely random hyperplanes might therefore in fact better divide the search space than before.

When using dimensionality reduction for LSH it is therefore clear that one should use the reduced data for the hashing part, which will reduce the time taken for hashing, and perhaps even improve the hashing. If one decides to reduce the data too much however, then dimensionality reduction might start having a negative effect instead, as more meaningful information is thrown out.

While dimensionality reduction might be beneficial for hashing, it could also reduce the accuracy of point comparisons, though exactly how much accuracy would be lost depends on how much information is thrown out which in turn depends on how much one is willing to reduce the dimensionality of the data.

One then has two choices if one wants to use dimensionality reduction for LSH. One can either reduce the data and use the reduced data both for hashing and for matching, or one can use it only for hashing and not for matching (in theory one can use it only for matching and not for hashing as well, though it is likely not beneficial to do so). It would really be a trade-off between accuracy and speed, and the trade-off might in some situations be worth it for hashing but not for matching.

Principal components as hyperplanes

For hyperplanes, it is generally beneficial that the points in the dataset are both somewhat equally distributed between the two sides of the hyperplane, and also that the distance of the points from the hyperplane is as large as possible.

As stated earlier, principal components are orthogonal vectors that point in the direction of maximum variance under the constraint that each principal component must be orthogonal to the principal components before it in the list. Before taking the principal components, mean normalization is performed.

What principal components achieve is therefore very similar to what one is trying to achieve when creating hyperplanes. The mean normalization first makes the points somewhat equally distributed on both sides of the origin in all dimensions. Finding the direction of maximum variance also means finding the direction in which the average squared distance of points to the hyperplanes is large. There is therefore reason to believe that principal components would make excellent hyperplanes.

One can store the mean of the training set separately so that the same vector can be subtracted from the base points and the query points. Then one can use the principal components as hyperplanes.

One disadvantage of using PCA to generate hyperplanes would be that one can only get one set of hyperplanes that way, this is a problem if one wants to use several tables.

There is also the case where some principal components will be much more important than others. To a certain extent, this is a strength, the most important principal components will then always be used, and the less important principal components are added later only if they are needed. However, it is also a problem, the first principal component might be much more significant than others down the line, yet they will all represent the same amount of bits in the hash code. One could try to remedy this by letting some hyperplanes represent more bits in the hash codes than others, but it would probably be more trouble than it's worth.

Finally, using principal components as hyperplanes doesn't actually bring any other functionality than what can easily be achieved by dimensionality reduction. What is achieved by using principal components as hyperplanes is the same as performing dimensionality reduction using those principal components, then setting the hyperplanes as splitting the space by whether points are positive or negative along each of these dimensions, then comparing the query points with the base points using their original form before dimensionality reduction. If one wants this functionality however, then using using principal components as hyperplanes is probably the more natural way of implementing it.

3.4.4 Lower Dimensional Hyperplanes

One can attempt to shorten the time spent on hashing by allowing hyperplanes to only exist in a subset of the 128 dimensions used for the feature descriptor vectors. Since the hyperplanes are 128 element arrays of floating point values this would mean using only a subset of those elements to represent the hyperplanes.

For example, one can use the first 16 bits for the first and ninth hyperplane, then the second 16 bits for the second and tenth hyperplanes, etc. Reducing the dimensionality of hyperplanes like this will reduce the amount of time required by the hashing phase since one no longer has to perform a vector multiplication in 128 dimensions (and thereby having to multiply 128 numbers to see which side of a hyperplane a point lands on). Instead, one would only need to multiple the relevant 16 elements of the feature vector with a 16 element hyperplane vector.

While reducing the dimensionality of the hyperplanes in this way does reduce the amount of computation needed for hashing, it also might reduce the effectiveness of the hashing. In general, since we are using random hyperplanes, we want to give the hyperplanes a free hand in terms of the possible forms they can take on, the hyperplanes splitting the feature space across a large multitude of dimensions is one of the strengths of the algorithm, as well as the hyperplanes being substantially different from each other both within and between LSH tables.

The randomness of the hyperplanes is more important when a large number of tables are used than when only a few are used, but there may be another important benefit of allowing the hyperplanes to take up all 128 dimensions. When a query point and its best match among the base points deviate from each other, it is reasonable to believe that they will differ more in some dimensions while remaining very similar in other dimensions; according to this logic, the more dimensions that are used for a hyperplane, the less sensitive the algorithm will be to random differences along only a small number of dimensions.

Part II

Implementation, Results, Discussion

Chapter 4

GPGPU Programming for LSH

4.1 Testing Environment

When testing this algorithm I will be using a publicly available dataset of SIFT feature descriptors. The dataset consists of 1 million base vectors, 10 000 query vectors, and a ground truth table containing the correct matches between the query vectors and the base vectors. The dataset also contains a learning set of 100 000 feature descriptors that can be used to train or tune the algorithm in any number of ways, in the case of LSH they can be used to fit hyperplanes i.e. precomputing the hyperplanes to partition the training data well in the hope that they will perform equally well on new data.

In order to cite this dataset I will cite the paper where the dataset was formally introduced in 2011 called "Product Quantization for Nearest Neighbor Search" [7], written by Hervé Jégou, Matthijs Douze and Cordelia Schmid.

For compiling the code I used the Portland Group Inc. (PGI) C++ compiler: `pgc++`. Specifically this version of the `pgc++` compiler is part of the NVHPC 21.5 package. Profiling was done mainly using NVIDIA's `nvprof` compiler. The "nvToolsExt" library was used to get some extra insight into the performance of particular parts of the program. The `pgc++` compiler often improves the code in various ways even without any OpenACC directives (particularly using SIMD instructions).

The CPU model that the program is tested on is "Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz", the GPU model is a "NVIDIA RTX2080Ti".

4.2 Main Phases of LSH Overview

Before showing how I optimized my implementation of LSH using OpenACC it is important that I cover some details about how the program works, in particular what parts of the program are computationally intensive, and also point out which of the computationally intensive parts are more suitable for GPGPU parallelization using OpenACC.

As mentioned in the introductory chapter, LSH consists of two major phases: constructing LSH tables for the base points, and matching the query points to the base points using those LSH tables. These two phases can further be split up into several steps, the resulting five stages of my LSH implementation are the following:

1. Calculate hash values for the base points.
2. Organize base vector references into LSH tables according to their hash values.
3. Calculate hash values for query points.
4. For each table, find the base points that had fallen into the same group as the query point and store them in a separate list (I named this stage as finding "potential matches" to the query points). This stage is mainly added to avoid duplicate comparisons, and to simplify stage 5.
5. Compare the query points to all their "potential matches", i.e. all the base points they got grouped with.

We need to calculate the hash values for both the base points and the query points, this is split into two different steps in the outline above, however they use the same function.

As mentioned earlier, when testing the implementation I will be using a dataset of 1 million base points and 10 000 query points. The hashing of the base points will therefore be significantly more computationally intensive than the hashing of the query points during testing, and the hashing of the query points will thus be largely negligible in comparison.

4.3 Function Explanation and Optimization

In this section of the thesis I will quickly present each function one by one, discuss to what extent they are parallelizable, and present parallelization and optimization of the two most time consuming and most parallelizable functions using OpenACC.

The two functions that will be parallelized using OpenACC are `calculate_hash_values` and `match_points`, used for calculating hash values and doing the raw point comparisons respectively. I will describe the steps I took to parallelize these two functions using OpenACC, and describe the speedup for each step. I will also quickly cover `organize_points_into_groups` and `find_potential_matches` since these functions provide insight to how the algorithm works

As covered in the section dedicated to OpenACC there are two types of OpenACC directives: the `parallel` directive, and the `kernels` directive. I decided to accelerate the program using the parallel directive as that gives more direct control over what happens in the program which makes it easier to document exactly what each step does since the program will not automatically make optimizations that I have not specified.

4.3.1 LSH Parameters

When testing the program, 32 LSH tables and 16 hyperplanes were used, there is no particular reason why the numbers were chosen to be powers of 2, but I chose these numbers since it gave decent results for the scale of the problem (and because they are *round* numbers), there may potentially be some advantage for the number of hyperplanes to be a multiple of eight due to how the hashing function ends up being parallelized, though from the results of different numbers of hyperplanes it does not seem to make much of a difference.

If one wants to make the time complexity of the algorithm strictly logarithmic, one technically has to set the number of hyperplanes equal to the log of the base points; however, the number of hyperplanes can really be any number, so I chose 16 as I found it to produce appropriate results. Using more hyperplanes should significantly reduce the search time, but it also reduces the chance of success and makes the algorithm require more memory for the LSH tables.

When running the algorithm with 32 LSH tables and 16 hyperplanes, the correct ratio (recall) was 83% and approximately 2.4% of the dataset was checked. These two results will not change as the program is accelerated using GPUs, the only thing that will change is the run-time.

4.3.2 Compiler Differences for the CPU Version

When compiling the program using the `g++` compiler with 32 LSH tables and 16 hyperplanes, the program used approximately 129 seconds for matching the entire dataset for the serial version.

When switching to the `pgc++` compiler the program uses 36.2 seconds instead. This is because the `pgc++` compiler, when given permission to do so, automatically parallelizes the application on the CPU side using SIMD instructions. The speed is increased by a factor of 3.56, meaning it is about three and a half times as fast.

The parts of the algorithm that become faster when changing to the `pgc++` compiler (which uses SIMD) are the parts of the algorithm that are most easily vectorized and able to be parallelized in a very fine-grained manner. This is the case for calculating hash values where the bottleneck mostly involve of vector multiplications, and it is also the case when doing raw point comparisons where the bottleneck mostly involve arithmetic operations on vectors (element-wise subtraction, multiplication, and accumulation).

The functions `calculate_hash_values` and `match_points` are thus the ones to increase in speed when the `pgc++` compiler is used, these are also the ones that I will parallelize using GPGPUs.

When switching from the `g++` compiler to the `pgc++` compiler the computation time of the `calculate_hash_values` decreases from 61.9 to 7.93 seconds, which is a 7.81x speedup; meanwhile, the computation time of the `match_points` function decreases from 64.6 to 26.0 seconds which results in a 2.38x speedup.

4.3.3 Calculating Hash Values

Original function

This is my function for calculating locality sensitive hash values for a single set of hyperplanes:

```
/**
 * Calculates the hash values of the input points given its
 * input hyperplanes, then stores those hash values into an
 * array called indexGroupMap which will later map the point
 * index into a group index used for indexing a second
 * array.
 *
 * @param vectorLength : Number of dimensions for each point
 * @param nPoints : Number of points to hash
 * @param points : The points to hash
 * @param nPlanes : Number of hyperplanes used
 * @param hyperplanes : Array of hyperplanes of
 *                      length nPlanes
 *
 * @param indexGroupMap : Output - Calculated hash values.
 *                         May be used as indices.
 */
void calculate_hash_values(
    // input
    int vectorLength,
    int nPoints, float* __restrict__ points,
    int nPlanes, float* __restrict__ hyperplanes,
    // output
    int* __restrict__ indexGroupMap)
{
    // - Calculate hash values,
    //   keep track of sizes of each group
    for (int i = 0; i < nPoints; i++) {

        float* point = &points[i * vectorLength];
        int hashcode = 0;// hashcode will be the group index

        // calculate hash value of the i'th point,
        // store resut in indexGroupMap
        for (int j = 0; j < nPlanes; j++) {
            float* hplane = &hyperplanes[j * vectorLength];
            // calculate point * hplane
            float vecMul = 0;
            for (int k = 0; k < vectorLength; k++) {
                vecMul += point[k] * hplane[k];
            }
            // set i'th bit to one if point is on
            // the "positive" side of hyperplane
            if (vecMul > 0) {
                hashcode = hashcode | (1 << j);
            }
        }
        indexGroupMap[i] = hashcode;  // save the hashcode
    }
}
```

This function will be called in a loop with different arrays of hyperplanes as input to calculate the hash values for each LSH table.

Using the g++ compiler, this function is able to hash all the base points in approximately 61.9 seconds in real time. Using the pgc++ compiler, the function was able to hash all the base points in about 7.93 seconds.

When parallelizing a program using GPGPU computing, what one is looking for are loops with very many independent (or partially independent) iterations. One also wants to avoid too much branching, complex behavior, per thread memory requirements, etc. Those program characteristics will make the program unsuitable for a single instruction multiple threads (SIMT) model; if there is too much branching, then each thread in a group will not be able to run concurrently; if each thread requires huge amounts of private memory, then the GPU will likely run out of memory.

The outer loop of the program iterates through each point in the point dataset. Then, for each point, the body of the loop calculates a hash code using the hyperplanes provided as arguments to the function.

In this particular program we can see that each iteration of the outer loop is completely independent. Meanwhile, there is little branching and no advanced function calls in the body of the loop. The outer loop should be easily parallelizable using GPGPU computing. I therefore start a parallel region at the beginning of this loop using the `parallel` OpenACC directive.

We will not stop here however, as there are more parallelizable parts of the program which should be indicated to the compiler.

The second loop in the program creates the hash value for the point. The hash value starts off as zero. The point's position in relation to each hyperplane is represented by a single bit in the hash variable, starting with the least significant bit representing the first hyperplane.

The loop iterates through each hyperplane. For every hyperplane, the vector multiplication between the hyperplane and the point is calculated. If the vector multiplication corresponds to a positive number, the corresponding bit in the hash variable is set to one, otherwise it remains zero.

One can thereby see that the second loop in the program is at least partially independent, each vector multiplication is carried out independently. When storing the result however, a *reduce* operation must be performed, since the result is not stored in its own independent value, but rather as a bit in an integer that is shared between each iteration of the loop, where the bits correspond to different iterations of the loop. For this type of reduce operation we can use the `reduce` clause in OpenACC; specifically, this loop can be parallelized using an *or-reduction*.

The innermost loop simply calculates the vector product of the hyperplane and the point. Since the loop is so short there may not be any benefit of splitting the work among several threads (at least if one is also parallelizing the middle loop). Still, one can inform the compiler of the parallelizability of the loop using a *plus-reduction*. Here we want to specify that it should run serially however (i.e. the loop should not be divided among threads). One can specify sequential execution of the loop using the `seq` clause. Since I wanted the loop to be serial I technically didn't have to specify any parallel behavior at all and the execution of the program would remain the same; I still did it in order to emphasize how one could parallelize the loop if one chooses to do so, the way it is written now means that if one were to simply remove the `seq` clause the compiler might try parallelize the loop if that is considered as advantageous for performance at compile time.

Instead of making the inner loop sequential, one can experiment with making the middle loop sequential and the inner loop parallel, or one could try to parallelize both loops, doing this will present different performance trade-offs. One possible disadvantage of parallelizing the middle loop like I did is that the number of workers may have to be fit to the number of hyperplanes. If the number of hyperplanes is 6 for example, and using 8 workers, two of the threads may be without work. This would not be a problem if one parallelized the inner loop instead since the number of dimensions in a SIFT feature descriptor will always be 128 dimensions; that solution might have other performance trade-offs so in some situations it could also be less effective. The code presented in this section however, is the one that I have written and tested, so those are the results I will present in this thesis.

Another piece of information that I provide for the compiler at this stage of optimization is memory movement between host and device. Even though one usually doesn't have to specify memory movement before optimizing the code further, in this case (and also in the case of the "match_points" function) I had to provide explicit memory movement operations from the very beginning, otherwise the compiler would not move the memory correctly and the program resulted in a run-time error. These problems seem to occur to varying degrees on different computers, I do not know the cause of this, though the data directives should be added for a finished optimization of the program, so this is of little consequence.

I used the `pcopyin` and `pcopyout` OpenACC data directives in order to move the memory regions between host and device before and after running the parallel region.

The memory that needs to be explicitly moved from host to device are the arrays used as input i.e. the '`points`' array and the '`hyperplanes`' array, these arrays contain input for the function, but they are not altered, so moving the memory back is not necessary.

'`indexGroupMap`' is the output array that must be moved from device to host. It stores group mappings for each point in '`points`'. Since the initial values are not used by the function the memory does not need to be moved from host to device.

Moving to GPU

Applying some basic OpenACC directives results in the following code.

```
// - Calculate hash values,
//   keep track of sizes of each group
#pragma acc data \
  pcopyin(points[nPoints*vectorLength]) \
  pcopyin(hyperplanes[nPlanes*vectorLength]) \
  pcopyout(indexGroupMap[nPoints])
{
    #pragma acc parallel loop
    for (int i = 0; i < nPoints; i++) {

        float* point = &points[i * vectorLength];
        int hashcode = 0;// hashcode will be the group index

        // calculate hash value of the i'th point,
        // store resut in indexGroupMap
        #pragma acc loop reduction(|:hashcode)
        for (int j = 0; j < nPlanes; j++) {
            float* hplane = &hyperplanes[j * vectorLength];
            // calculate point * hplane
            float vecMul = 0;
            #pragma acc loop reduction(+:vecMul) seq
            for (int k = 0; k < vectorLength; k++) {
                vecMul += point[k] * hplane[k];
            }
            // set i'th bit to one if point is on
            // the "positive" side of hyperplane
            if (vecMul > 0) {
                hashcode = hashcode | (1 << j);
            }
        }
        indexGroupMap[i] = hashcode;  // save the hashcode
    }
}
```

After moving computation from the GPU to the CPU, the new time of the program is 4.19s resulting in a 1.89x (89%) speedup from the CPU version compiled with `pgc++`.

The program now runs on the GPU, but how the work is delegated among threads is still unclear. We did not specify, for example, how many threads should be used for the reduction of the hash code (the middle loop). With the directives given we could have one thread executing all the code for every point (a very coarse-grained approach), or we can have up to "`nPlanes`" number of threads working together to find hash codes for each point (a more fine-grained approach).

OpenACC splits the threads into workers and vectors, the workers do the more coarse-grained work, whereas the vectors do the more fine-grained work. In OpenACC we can easily specify these values using the *num_workers* and *vector_length* clauses. *num_workers* multiplied by *vector_length* can be no higher than the highest possible number of threads in a CUDA block, for the GPU I'm using for this thesis that limit is 1024.

It is difficult to know exactly what division of labor should take place to achieve good results. The good thing about OpenACC is that one can easily experiment with these values without changing the code substantially. All one has to do is change the values between number of workers and number of vectors until one finds a balance that performs well.

Loop Optimization

```
// - Calculate hash values,
//   keep track of sizes of each group
#pragma acc data \
  pcopyin(points[nPoints*vectorLength]) \
  pcopyin(hyperplanes[nPlanes*vectorLength]) \
  pcopyout(indexGroupMap[nPoints])
{
    #pragma acc parallel loop gang worker \
      num_workers(128) vector_length(8)
    for (int i = 0; i < nPoints; i++) {

        float* point = &points[i * vectorLength];
        int hashcode = 0;// hashcode will be the group index

        // calculate hash value of the i'th point,
        // store resut in indexGroupMap
        #pragma acc loop reduction(|:hashcode) vector
        for (int j = 0; j < nPlanes; j++) {
            float* hplane = &hyperplanes[j * vectorLength];
            // calculate point * hplane
            float vecMul = 0;
            #pragma acc loop reduction(+:vecMul) seq
            for (int k = 0; k < vectorLength; k++) {
                vecMul += point[k] * hplane[k];
            }
            // set i'th bit to one if point is on
            // the "positive" side of hyperplane
            if (vecMul > 0) {
                hashcode = hashcode | (1 << j);
            }
        }
        indexGroupMap[i] = hashcode;  // save the hashcode
    }
}
```

After optimizing the loops, the new time of the program is 3.68s resulting in a 1.14x (14%) speedup from the last stage of optimization, a 2.15x total speedup from the CPU version compiled with `pgc++` (utilizing SIMD), and a 16.8x speedup from the CPU version compiled with `g++`.

The function copies the '`hyperplanes`' and '`points`' arrays from the host to device for each call to the function. Similarly the '`indexGroupMap`' array is copied from the device to host for each time the function is called. Since this function is called in a loop for each LSH table by another calling function, this will trigger a memory movement phase for every iteration in the loop of the calling function, i.e. there will be a memory movement phase for each LSH table.

Starting and stopping the memory movement operations requires large amounts of overhead. An interesting fact however, is that in my implementation of LSH these arrays are stored contiguously in memory, this makes it very simple to move all this memory in bulk instead, assuming the device has enough memory for all the points. There seems to be a benefit of moving these arrays from host to device in their entirety, then iteratively calling the function to operate on them, and finally, after that is done, moving the results back in their entirety, this seems to avoid a lot of overhead associated with memory movement.

Here is some profiling information for the hashing phase in the current state of the program[1]:

```
NVTX result:
Range "Calculating hash values for base vectors"
Type    Time(%)      Time   Calls    Name
Range:
        100.00%  3.50159s       1   Calculating hash values
GPU activities:
        76.64%  1.33689s     1024   [CUDA memcpy HtoD]
        22.80%  397.67ms       32   calculate_hash_values_gpu
        0.57%   9.9205ms       32   [CUDA memcpy DtoH]
API calls:
        86.62%  8.0815ms     1024   cuMemcpyHtoDAsync
        10.38%  968.24us       32   cuLaunchKernel
        3.00%   280.12us       32   cuMemcpyDtoHAsync
```

In addition to the time spent directly on GPU computation and memory movement there is also various other types of overhead. Here is information of the most time consuming of those types of overhead taken from the profile of the entire program:

```
Type    Time(%)   Time       Calls    Name
OpenACC (excl):
        71.23%    2.67870s      64    acc_enter_data@lsh_parallel_5.cpp:630
        12.17%    457.56ms     128    acc_wait@lsh_parallel_5.cpp:630
        8.35%     314.14ms       2    acc_wait@lsh_parallel_5.cpp:887
        6.53%     245.73ms       1    acc_enter_data@lsh_parallel_5.cpp:887
        1.26%     47.387ms      64    acc_wait@lsh_parallel_5.cpp:654
```

The hash function is located around line 630, therefore the first and second line is the OpenACC overhead corresponding to the hash function. At this point in optimization, this overhead is very large.

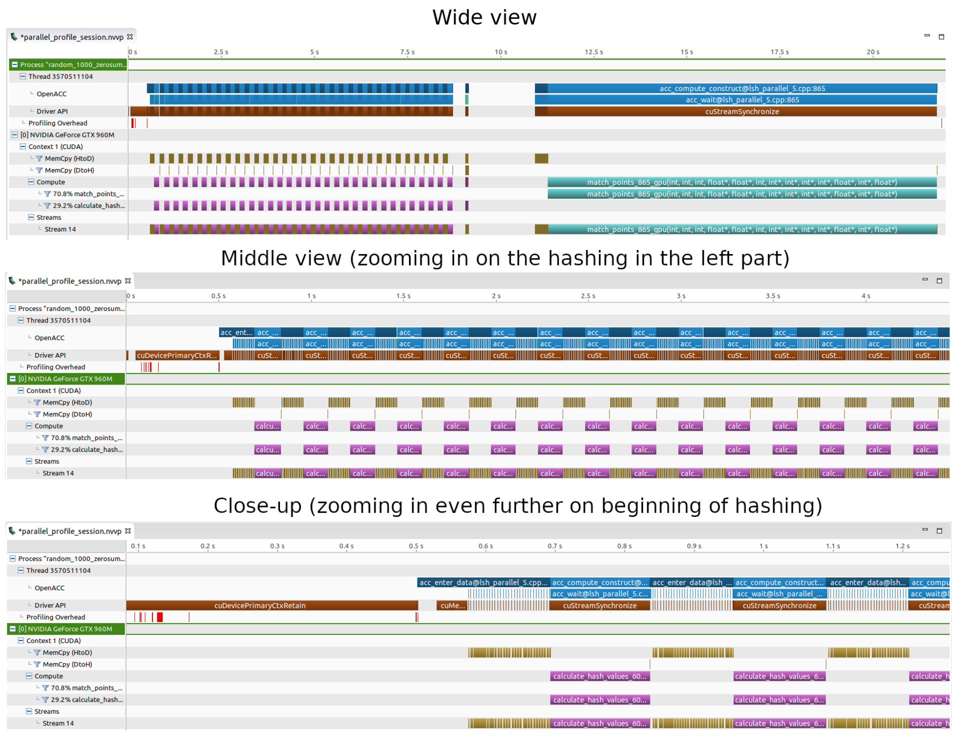
As we can see, an enormous portion of the program is spent on moving memory between host and device. The cause of this can be seen in Figure 4.1.

In Figure 4.1 we can see that the point matching is executed in a very orderly fashion: first, required memory is moved in; then,

[1]The profiling information is significantly shortened so that it has space to be presented here

the `match_points_gpu` kernel is launched and executes without any disruption. The memory is moved between host and device for each time `match_points` is called, this is effective largely because `match_points` is only called a single time.

The `calculate_hash_values` function on the other hand is called by `calculate_indexGroupMap` repeatedly, for each table, in a loop. This leads to what we see in Figure 4.1 where phases of calculating hash values are interspersed by phases where memory is moved. Changing between phases of moving memory and performing calculations on the GPU is bad for performance, and this is therefore something we want to avoid.



Wide view
Middle view (zooming in on the hashing in the left part)
Close-up (zooming in even further on beginning of hashing)
*parallel_profile_session.nvvp
Process "random_1000_zerosum...
Thread 3570511104
OpenACC
Driver API
Profiling Overhead
[0] NVIDIA GeForce GTX 960M
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
70.8% match_points_...
29.2% calculate_hash...
Streams
Stream 14
acc_compute_construct@lsh_parallel_5.cpp:865
acc_wait@lsh_parallel_5.cpp:865
cuStreamSynchronize
match_points_865_gpu(int, int, int, float*, float*, int, int*, int*, int*, int*, float*, int*, float*)
cuDevicePrimaryCtxRetain
acc_enter_data@lsh_parallel_5.cpp...
calculate_hash_values_60...

Figure 4.1: Figure showing a visual profile of the LSH algorithm where memory is moved before each call to `calculate_hash_values`. The top part shows a "wide view" of the program, with point matching on the right side and hashing on the left side. The middle part zooms in on the hashing. The "close-up" zooms in even further on the hashing in the beginning of the program. (Visual profiling is done on a different computer with some different parameters, so the run-times are not relevant)

Moving Memory in Bulk

As suggested earlier, instead of doing many smaller memory movement instructions, it turns out it is more effective to do a single large memory movement instruction before and after all the iterations of the function. The following code snippet shows how this is done by adding/changing OpenACC data directives for two functions:

```
/**
 * Calculates indexGroup for all tables by
 * calling 'calculate_hash_values' int a loop.
 *
 * @param vectorLength : Number of dimensions for
 *                       each point.
 * @param numTables : Number of LSH tables.
 * @param nPoints : Number of points to hash.
 * @param points : The points to hash.
 * @param nPlanes : Number of hyperplanes used
 * @param hyperplanes : Array of hyperplanes of length
 *                      nPlanes
 * @param indexGroupMapTableLen : The length of a single
 *                                table in indexGroupMap
 *
 * @param indexGroupMap : Output.
 *                        Calculated hash values,
 *                        may be used as indices
 */
void calculate_indexGroupMap(
    // input
    int vectorLength, int numTables,
    int nPoints, float* __restrict__ points,
    int nPlanes, float* __restrict__ hyperplanes,
    int indexGroupMapTableLen,
    // output
    int* __restrict__ indexGroupMap)
{
#pragma acc data \
  pcopyin(points[nPoints*vectorLength]) \
  pcopyin(hyperplanes[numTables*nPlanes*vectorLength]) \
  pcopyout(indexGroupMap[numTables*nPoints])
 {
    const int hyperplanesTableLen = nPlanes * vectorLength;
    for (int table = 0; table < numTables; table++) {
        float* hyperplanes2 =
            hyperplanes + table * hyperplanesTableLen;
        int* indexGroupMap2 =
            indexGroupMap + table * indexGroupMapTableLen;

        // - Match points -
        calculate_hash_values(
            vectorLength, nPoints, points,
            nPlanes, hyperplanes2, indexGroupMap2);
    }
 }
}
```

```
void calculate_hash_values(
    // input
    int vectorLength,
    int nPoints, float* __restrict__ points,
    int nPlanes, float* __restrict__ hyperplanes,
    // output
    int* __restrict__ indexGroupMap)
{
// - Calculate hash values,
//   keep track of sizes of each group
#pragma acc data \
  present(points[nPoints*vectorLength]) \
  present(hyperplanes[nPlanes*vectorLength]) \
  present(indexGroupMap[nPoints])
 {
    #pragma acc parallel loop \
      num_workers(128) vector_length(8)
    for (int i = 0; i < nPoints; i++) {

        float* point = &points[i * vectorLength];
        int hashcode = 0;// hashcode will be the group index

        // calculate hash value of the i'th point,
        // store resut in indexGroupMap
        #pragma acc loop reduction(|:hashcode)
        for (int j = 0; j < nPlanes; j++) {
            float* hplane = &hyperplanes[j * vectorLength];
            // calculate point * hplane
            float vecMul = 0;
            #pragma acc loop reduction(+:vecMul) seq
            for (int k = 0; k < vectorLength; k++) {
                vecMul += point[k] * hplane[k];
            }
            // set i'th bit to one if point is on
            // the "positive" side of hyperplane
            if (vecMul > 0) {
                hashcode = hashcode | (1 << j);
            }
        }
        indexGroupMap[i] = hashcode;  // save the hashcode
    }
 }
}
```

`calculate_indexGroupMap` is the function that calls the function `calculate_hash_values` for each table. In order to reduce the number of times the program must perform memory movement, the memory movement directives are moved from the `calculate_hash_values` function to the `calculate_indexGroupMap` function.

In `calculate_hash_values` the memory areas are instead marked as present using the `present` directive. This makes it clear to the compiler that the memory addresses are already copied over to the device, and the memory pointers in question will point to device memory at the execution of this function.

Optimizing data locality by moving memory in bulk increases the performance of the function substantially. The time goes from 3.68 seconds in the last optimization to 0.726 seconds. That is a 5.07x incremental speedup from the last optimization, a 10.9x total speedup from the CPU version compiled with pgc++ (which utilizes SIMD), and an 85.3x speedup from the serial CPU version compiled with g++.

We can also see from the profile that the time spent on memory movement has been significantly reduced:

```
NVTX result:
Range "Calculating hash values for base vectors"
Type    Time(%)      Time   Calls    Name
Range:
        100.00%  934.81ms       1   Calculating hash values
GPU activities:
        89.09%  422.75ms       32   calculate_hash_values_gpu
        8.86%  42.023ms       32   [CUDA memcpy HtoD]
        2.06%  9.7546ms       32   [CUDA memcpy DtoH]
API calls:
        55.45%  380.47us     1024   cuMemcpyHtoDAsync
        26.44%  181.43us       32   cuLaunchKernel
        18.12%  124.31us       32   cuMemcpyDtoHAsync
```

The time spent on memory movement used to be approximately 1.3 seconds, now it is approximately 0.042 seconds.

We can also see that the time spent on OpenACC overhead throughout the entire program has gone significantly down from what it was before:

```
Type    Time(%)    Time       Calls    Name
OpenACC (excl):
        36.73%    428.69ms      64    acc_wait@lsh_parallel_6.cpp:632
        27.53%    321.39ms       2    acc_wait@lsh_parallel_6.cpp:895
        21.13%    246.66ms       1    acc_enter_data@lsh_parallel_6.cpp:895
        10.17%    118.73ms       2    acc_enter_data@lsh_parallel_6.cpp:712
        3.54%     41.275ms       2    acc_exit_data@lsh_parallel_6.cpp:712
```

The time spent moving data in the hashing function has now decreased substantially. The data movement directives have now been moved to line 712, which is inside the `calculate_indexGroupMap`. We can see that the pure overhead related to hashing went from being the largest contributing factor of OpenACC overhead to being the third largest contributor, and has been reduced from taking approximately 2.68 seconds to 0.247 seconds.

And finally we can see that from Figure 4.2 that the program no longer alternates between memory movement and calculation during calculation of the hash values; instead, memory movement is done in bulk before and after execution which significantly reduces the amount of overhead involved in memory movement and shortens the run-time of the program.

Limitation of OpenACC

One limitation of OpenACC is somewhat noticeable when optimizing the hashing function. In the *loop optimization* stage I experimented with different ways to delegate work among gangs, workers and vectors, and I

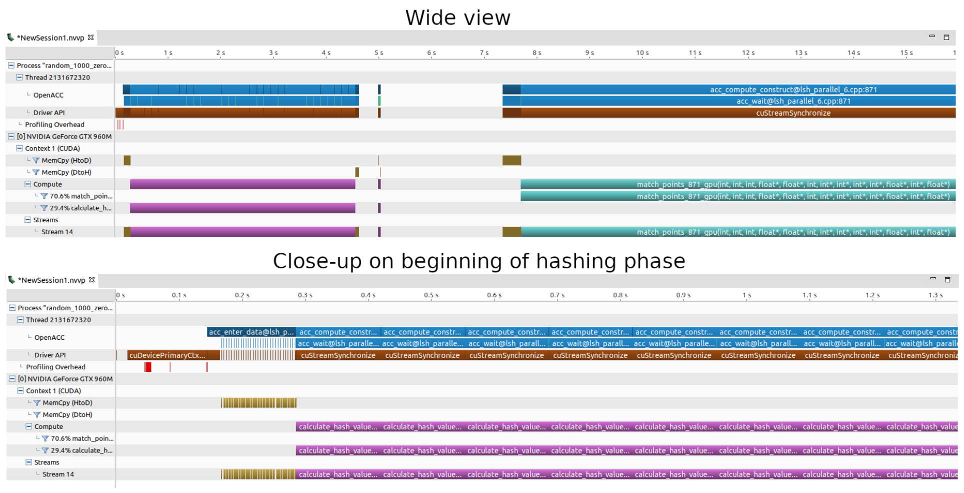


Figure 4.2: Figure showing a visual profile of the LSH algorithm where memory is moved before each call to `calculate_hash_values`. The top part shows a "wide view" of the program, with point matching on the right side and hashing on the left side. The "close-up" zooms in on the hashing in the beginning of the program. (Visual profiling is done on a different computer with some different parameters, so the run-times are not relevant)

decided it was best to divide the outer loop among both gangs and workers. As defined in this function by the directive "`num_workers(128)`": for every gang there should typically be 128 workers, each worker handling one iteration of the outer loop. These workers will iterate through the exact same hyperplanes.

In the CUDA programming model, threads in a block should be able to share memory. Ideally, for the hashing function, one would want the threads to be able to load all the hyperplanes into shared memory, so they can share the memory and use less space and lower the number of load operations for each thread; OpenACC does not have a way to do this. OpenACC does include a `cache` directive for utilizing shared memory among workers and vectors; however, it can only be used in a more fine-grained fashion, and to my knowledge it can not be used to solve this type of problem.

4.3.4 Constructing LSH Tables

Another major part of the LSH algorithm is the construction of the LSH tables itself. Though this part of the algorithm was itself not optimized using OpenACC, it might be worth covering here as it gives perspectives to how the other functions were optimized.

The function does not add the length as the last element of the *groupIndexMap* array, but that functionality can very easily be added.

There may be some performance gain in using an optimized library for the exclusive scan of *groupSizeMap*, but it makes little difference in this case, since the exclusive scan is only a very small part of an already not very time consuming function.

```
/**
 * Takes as input an array of point-index to group mappings.
 * Turns this into a data structure which takes the hash
 * value of a point and returns a group index into a new
 * array called groupArray, which itself contains indices
 * into the original point array.
 * The arrays, groupSizeMap and groupIndexMap, are used
 * to easily index or iterate through specific groups
 * in the groupArray. This is handy when we want to
 * compare a point to other points in its group.
 *
 * @param nPoints : The size of indexGroupMap, groupArray,
 *                  and in general the number of points

 * @param nGroups : The number of groups.
 *
 * @param indexGroupMap :  Mapping from indices to groups,
                           calculated by function
 *                         'calculate_hash_values'
 *
 * @param groupIndexMapTails : Temporary array used when
 *                             creating groupArray.
 *                             Preallocated only for possible
 *                             increased efficiency.
 *
 * @param groupArray : Output.
 *                     An array with point indices sorted
 *                     by their group.
 *                     Maps into the original point array.
 *
 * @param groupSizeMap : Output. Array of group sizes
 *
 * @param groupIndexMap : Output.
 *                        Array of group indices, indexes
 *                        groupArray
 */
void organize_points_into_groups(
    // input
    int nPoints, int nGroups,
    int* __restrict__ indexGroupMap,
    // preallocated temporary storage
    int* __restrict__ groupIndexMapTails,
    // output
    int* __restrict__ groupArray,
    int* __restrict__ groupSizeMap,
    int* __restrict__ groupIndexMap)
{
    // find the size of each group
    memset(groupSizeMap, 0, nGroups * sizeof(int));
    for (int i = 0; i < nPoints; i++) {
        groupSizeMap[indexGroupMap[i]]++;
    }
    // Find group indices into the group array 'groupArray'
    // This is really just an exclusive scan
    int cnt = 0;
    for (int i = 0; i < nGroups; i++) {
        groupIndexMap[i] = cnt;
        groupIndexMapTails[i] = cnt;
        cnt += groupSizeMap[i];
    }

    // Fill groupArray
    for (int i = 0; i < nPoints; i++) {
        // what group did the point get hashed into
        int hashcode = indexGroupMap[i];
        // start index of that group in groupArray
        //  + number of elements currently inserted
        int idx = groupIndexMapTails[hashcode]++;
        // add the point to the group
        groupArray[idx] = i;
    }
}
```

The logic behind this algorithm was described briefly in words in the introductory LSH section, Section 3.3.9. It is perhaps easier to understand by simply looking at the code however.

Note that here we are in a way sorting the indices by their hash codes, and combining points with equal hash codes into a single group. We are doing this in linear time, this is possible since we are exploiting the fact that the hashcodes can only take on values within a limited range with a clear upper limit equal to $2^H - 1$ where H is the number of hyperplanes, and a lower limit of 0.

4.3.5 Finding Potential Matches

When beginning the matching phase of the algorithm, where matches for query points must be looked for in an LSH table already created, the first thing that must be done is hashing the query points.

The optimization of the hashing algorithm has already been covered. I will therefore move to the next step, which consists of finding the box associated with the query point for each LSH table and look for potential candidates for the best match among the base points located in these boxes. If a base point appears inside one of these boxes it will be added to a separate set containing all the potential best-match candidates for the current query point.

The algorithm uses a large array to keep track of whether or not a base point is already represented in the set of potential matches for this query point; if it is, then the algorithm simply moves on; if it is not, then the point index is added to the set. This is done to avoid duplicate point comparisons within the raw comparison phase of the program.

The final number of potential matches for all query points is unknown, thus the final length of the *potentialMatches* array is unknown. In this program, that problem is solved by simply allocating a very large amount of memory and quitting the program if more memory is needed. Therefore, if one tries to run the program in a case where all the query points have a large amount of potential matches each such that the total number of potential matches exceeds this maximum limit, then the program will simply print an error message and terminate.

One can experiment with other solutions to this issue: one could use an array that expands if the number of potential is larger than expected, perhaps a solution that even stores parts of the array in secondary memory if there is not enough memory in primary memory (though this would make the algorithm slower if this event triggers). For this implementation I decided to just quit the program in case the program runs out of memory, as that is a simpler solution that satisfies my goals with the implementation for this thesis.

```
/**
 * For every point, this function goes through all the
 * groups in groupArray it can get mapped into.
 * groupArray stores these points as indices into the
 * original point array; it does this for all tables.
 * Then the function puts indices it can get mapped to into
 * that point's place in the potentialMatches array. After
 * that, the function creates two other arrays called
 * potentialMatchesIndices and potentialMatchesLengths,
 * these two arrays are used to index and iterate the
 * potentialMatches array.
 *
 * @param numTables : Number of LSH tables
 * @param nPoints1 : Number of base points and size of
 *                   groupArray
 * @param nPoints2 : Number of query points
 * @param indexGroupMap : Calculated hash values for points2
 * @param indexGroupMapTableLen : Length of indexGroupMap
 *                                per table
 * @param groupArray : Arrays with point indices sorted by
 *                     their group.
 * @param groupSizeMap : Arrays with sizes of each group in
 *                       groupArray.
 * @param groupIndexMap : Arrays with group indices,
 *                        indexes into groupArray.
 * @param groupMapTableLen : Length of all the groupMaps,
 *                           (number of groups)
 * @param potentialMatchesMaxLen : The current max size of
 *                                 potentialMatchesLen
 *
 * @param potentialMatches : Output.
 *        An array with indices into point1 containing the
 *        indices of all possible matches for each point
 *        in points2
 *
 * @param potentialMatchesIndices : Output.
 *        Indices for potentialMatches.
 *
 * @param potentialMatchesLengths : Output.
 *        Number of matches for each point in point2.
 *        Used to iterate through potentialMatches.
 *
 * @return The total amount of matches
 */
int find_potential_matches(
    // inputs
    int numTables,
    int nPoints1,
    int nPoints2,
    int* __restrict__ indexGroupMap,
    int indexGroupMapTableLen,
    int* __restrict__ groupArray,
    int* __restrict__ groupSizeMap,
    int* __restrict__ groupIndexMap,
    int groupMapTableLen,
    int potentialMatchesMaxLen,
    // outputs
    int** __restrict__ potentialMatches,
    int* __restrict__ potentialMatchesIndices,
    int* __restrict__ potentialMatchesLengths)
{

const int groupArrayTableLen = nPoints1;

int* checkedArr = (int*)malloc(nPoints1 * sizeof(int));
memset(checkedArr, -1, nPoints1 * sizeof(int));
int totalMatchCount = 0;

// find possible matches
for (int i = 0; i < nPoints2; i++)
 {
    potentialMatchesIndices[i] = totalMatchCount;

    for (int table = 0; table < numTables; table++) {
        int* indexGroupMap2 = indexGroupMap
            + table * indexGroupMapTableLen;
        int* groupSizeMap2 = groupSizeMap
            + table * groupMapTableLen;
        int* groupIndexMap2 = groupIndexMap
            + table * groupMapTableLen;
        int* groupArray2 = groupArray
            + table * groupArrayTableLen;

        // Find the group of elements from
        // groupArray to match with
        int hashcode = indexGroupMap2[i];
        int size = groupSizeMap2[hashcode];
        int startIdx = groupIndexMap2[hashcode];

        // Add points to potentialMatches
        for (int j = startIdx; j < startIdx + size; j++) {
            int idx = groupArray2[j];
            if (checkedArr[idx] != i) {
                checkedArr[idx] = i;

                if (totalMatchCount>=potentialMatchesMaxLen)
                {
                    printf("Not␣enough␣space\n");
                    exit(1);
                }

                (*potentialMatches)[totalMatchCount] = idx;
                totalMatchCount++;
            }
        }
    }

    potentialMatchesLengths[i] =
        totalMatchCount - potentialMatchesIndices[i];
}

free(checkedArr);

return totalMatchCount;

}
```

This function was not parallelized on GPUs using OpenACC, and I will outline a few reasons why.

- The program uses a very large amount of memory for a lookup table to avoid duplicate point comparisons down the line. Such a large table (or even a smaller bitmap) could not be maintained for every single thread running on a GPU. This feature would therefore either have to be implemented another way or abandoned.
- The upper bound of possible matches for a single query point should be very high, and the exact amount of memory needed for each point is not known beforehand. Due to the law of large numbers it should be easier to find a stable upper limit when matching the entire dataset, since the total number of potential matches is more stable (relative to its size) than the number of potential matches for individual query points.
- Very little actual computation takes place in this part of the program, it mostly consists of reading and writing from memory.

While parallelizing `find_potential_matches` using GPUs is difficult due to the reasons I outlined above, there might still be benefits to parallelizing the function using multiple CPU cores or multiple CPUs completely (since CPUs are considered to be generally better at logic tasks and have more memory).

Each CPU thread could check the tables for potential matches for their assigned sections of the query set. Then after all CPUs are done, the potential matches tables for each CPU can be combined into a single table (or each of the CPU threads could run a matching routine for their own table, that is also a possible solution). This would be a coarse-grained approach to parallelization.

4.3.6 Matching Points

Next I will present the function for actually matching the points. This function consists of going through each query point, looking up which base points they should be compared with using the `potentialMatches` table, then doing the raw point comparisons and saving the best and second best match.

Original function

```
/**
 * Uses results from find_potential_matches to match points1
 * and points2
 *
 * @param vectorLength : Number of dimensions for each point
 *
 * @param numTables : Number of LSH tables.
 *
 * @param nPoints1 : Number of points in points1.
 *
 * @param nPoints2 : Number of points in points2.
 *
 * @param points1 : Array of points used to create the
 *                  LSH tables. (base points)
 *
 * @param points2 : Array of points to be matched with
 *                  points1. (query points)
 *
 * @param potentialMatches : An array with indices into
 *              point1 containing the indices of all
 *              possible matches for each query point
 *              in points2.
 *              (i.e. potential matches between points2
 *               and points1)
 *
 * @param potentialMatchesIndices : Indices used to index
 *                                  into potentialMatches.
 *
 * @param potentialMatchesLengths : Number of matches
 *              for each point in points2.
 *              Used to iterate through potentialMatches.
 *
 * @param lshMatches : Output.
 *          Index into points1 representing the best match
 *          between points2  and points1
 *
 * @param bestMatchDists : Output.
 *              Distances between points for every match in
 *              lshMatches.
 *
 * @param lshMatches2 : Output.
 *              Index into points1 representing the 2nd
 *              best match between points2  and points1.
 *
 * @param bestMatchDists2 : Output.
 *               Distances between points for every match
 *               in lshMatches2
 */
void match_points(
    // inputs
    int vectorLength, int nPoints1, int nPoints2,
    float* __restrict__ points1,
    float* __restrict__ points2,
    int nPotentialMatches,
    int* __restrict__ potentialMatches,
    int* __restrict__ potentialMatchesIndices,
    int* __restrict__ potentialMatchesLengths,
    // outputs
    int* __restrict__ lshMatches,
    float* __restrict__ bestMatchDists,
    int* __restrict__ lshMatches2,
    float* __restrict__ bestMatchDists2)
{
    for (int i = 0; i < nPoints2; i++) {
        bool changed = false;
        float bestMatchDist = bestMatchDists[i];
        float bestMatchDist2 = bestMatchDist;
        int match = -1;
        int match2 = -1;
        int jStart = potentialMatchesIndices[i];
        int jEnd = potentialMatchesIndices[i]
                   + potentialMatchesLengths[i];

        // Match the points
        for (int j = jStart; j < jEnd; j++) {
            int idx = potentialMatches[j];
            //diff = sum((points2[i][:]-points1[idx][:]).^2)
            float diff = 0;
            for (int k = 0; k < vectorLength; k++) {
                float tmp = points2[i * vectorLength + k]
                        - points1[idx * vectorLength + k];
                diff += tmp * tmp;
            }
            // check if distance is lowest distance so far
            if (diff < bestMatchDist) {
                // save old value as second best
                bestMatchDist2 = bestMatchDist;
                match2 = match;
                // update best value
                bestMatchDist = diff;
                match = idx;
                changed = true;
            }
        }

        if (changed) {
            lshMatches[i] = match;
            bestMatchDists[i] = bestMatchDist;
            lshMatches2[i] = match2;
            bestMatchDists2[i] = bestMatchDist2;
        }
    }
}
```

Using the g++ compiler this function used 64.6s to execute for the testing data set. When using the pgc++ compiler (which speeds up parts of the program using SIMD instructions) the time taken was 26.0s, which is roughly a 2.48x speedup from the g++ compiled version.

Moving to GPU

Like when optimizing `calculate_hash_values`, the first thing we need to do is recognize *for-loops* where the iterations of the loop are independent from each other.

The outer loop finds the best and second best match for each query point, since the iterations of this loop are independent from each other we can simply start a parallel region there using the `parallel` and `loop` directives.

The second loop iterates through the potential matches for the current point, it calculates the differences between the point and each of its potential matches, and stores the index of the best and second best match separately.

In this particular loop the iterations are dependent on each other. If this loop were to be parallelized one would need to reduce the best and second best match with respect to the difference value. This is an advanced reduction and to my knowledge there is no straight forward way to express this reduction in OpenACC, therefore the second loop itself cannot be parallelized using OpenACC; different sections inside body of the loop can still be parallelized regardless. Another issue with parallelizing this loop is that the number of iterations will be highly variable, thus there is no easy way to find a good balance of threads to use in this loop in relation to other loops.

The third loop however, the innermost loop, can be parallelized. This loop simply finds the squared euclidean distance between two points using vector subtraction and accumulating the squared elements. This loop can be reduced by using the `reduction` clause, in full: `reduction(+:diff)`.

```
#pragma acc data \
    pcopyin(points1[nPoints1*vectorLength]) \
    pcopyin(points2[nPoints2*vectorLength]) \
    pcopyin(potentialMatches[nPotentialMatches])\
    pcopyin(potentialMatchesIndices[nPoints2]) \
    pcopyin(potentialMatchesLengths[nPoints2]) \
    pcopy(lshMatches[nPoints2]) \
    pcopy(bestMatchDists[nPoints2]) \
    pcopy(lshMatches2[nPoints2]) \
    pcopy(bestMatchDists2[nPoints2])
{
    #pragma acc parallel loop
    for (int i = 0; i < nPoints2; i++) {
        bool changed = false;
        float bestMatchDist = bestMatchDists[i];
        float bestMatchDist2 = bestMatchDist;
        int match = -1;
        int match2 = -1;
        int jStart = potentialMatchesIndices[i];
        int jEnd = potentialMatchesIndices[i]
                   + potentialMatchesLengths[i];

        // Match the points
        for (int j = jStart; j < jEnd; j++) {
            int idx = potentialMatches[j];
            //diff = sum((points2[i][:]-points1[idx][:]).^2)
            float diff = 0;
            #pragma acc loop reduction(+:diff)
            for (int k = 0; k < vectorLength; k++) {
                float tmp = points2[i * vectorLength + k]
                        - points1[idx * vectorLength + k];
                diff += tmp * tmp;
            }
            // check if distance is lowest distance so far
            if (diff < bestMatchDist) {
                // save old value as second best
                bestMatchDist2 = bestMatchDist;
                match2 = match;
                // update best value
                bestMatchDist = diff;
                match = idx;
                changed = true;
            }
        }

        if (changed) {
            lshMatches[i] = match;
            bestMatchDists[i] = bestMatchDist;
            lshMatches2[i] = match2;
            bestMatchDists2[i] = bestMatchDist2;
        }
    }
}
```

After moving computation from the CPU to the GPU the function uses 1.01s during the execution of the program. This is a speedup of 25.7x speedup from the CPU version compiled using pgc++, and a 64.0x speedup from the fully serial CPU version compiled with g++.

Loop Optimization

Again, the program can be further optimized by explicitly specifying how many threads should perform each part of the program. GPGPUs tend to work best when the number of threads in a kernel is some multiple of 32; I therefore used an OpenACC vector size of 32, i.e. the innermost loop is always split among 32 threads. The vector size is not larger than 32 so that as many kernels as possible can run concurrently.

```
#pragma acc data \
    pcopyin(points1[nPoints1*vectorLength]) \
    pcopyin(points2[nPoints2*vectorLength]) \
    pcopyin(potentialMatches[nPotentialMatches])\
    pcopyin(potentialMatchesIndices[nPoints2]) \
    pcopyin(potentialMatchesLengths[nPoints2]) \
    pcopy(lshMatches[nPoints2]) \
    pcopy(bestMatchDists[nPoints2]) \
    pcopy(lshMatches2[nPoints2]) \
    pcopy(bestMatchDists2[nPoints2])
{
    #pragma acc parallel loop gang worker \
        num_workers(1) vector_length(32)
    for (int i = 0; i < nPoints2; i++) {
        bool changed = false;
        float bestMatchDist = bestMatchDists[i];
        float bestMatchDist2 = bestMatchDist;
        int match = -1;
        int match2 = -1;
        int jStart = potentialMatchesIndices[i];
        int jEnd = potentialMatchesIndices[i]
                   + potentialMatchesLengths[i];

        // Match the points
        for (int j = jStart; j < jEnd; j++) {
            int idx = potentialMatches[j];
            //diff = sum((points2[i][:]-points1[idx][:]).^2)
            float diff = 0;
            #pragma acc loop reduction(+:diff) vector
            for (int k = 0; k < vectorLength; k++) {
                float tmp = points2[i * vectorLength + k]
                        - points1[idx * vectorLength + k];
                diff += tmp * tmp;
            }
            // check if distance is lowest distance so far
            if (diff < bestMatchDist) {
                // save old value as second best
                bestMatchDist2 = bestMatchDist;
                match2 = match;
                // update best value
                bestMatchDist = diff;
                match = idx;
                changed = true;
            }
        }
        if (changed) {
            lshMatches[i] = match;
            bestMatchDists[i] = bestMatchDist;
            lshMatches2[i] = match2;
            bestMatchDists2[i] = bestMatchDist2;
        }
    }
}
```

After optimizing the loops the function uses 0.584s during execution of the program, resulting in a 44.5x speedup from the CPU version compiled with the pgc++ compiler, a 110.6x speedup from the serial CPU version compiled with the g++ compiler, and a 1.73x iterative speedup from the last optimization stage where the function executes on the GPU without loop optimization.

There is a similar version of this function that is optimized in Section 6.7.3, that one can compare and contrast with this one. The two functions have roughly the same behavior, but are implemented and optimized slightly differently. This is mostly just due to certain design choices I made when creating this implementation that I tried to change when I had the chance to re-implement the function. The differences are minor however, and they are almost the same function.

Chapter 5

Normal (multi-table) LSH: Results

5.1 Performance of Each Optimization

First, I will present some statistics that are the same for all optimization stages of the program.

Average portion of search space searched is the average percentage of the number of base points that had to be searched for each query point. For the testing set and hyperplanes I am using, that portion of search space is at about 2.4%.

The correct classification ratio as I called it can also just be called the recall of the program. It is the percentage of all best matches made that were correct. For the testing set and hyperplanes I am using, that portion of matches that were correctly classified is roughly 83%.

The relation between the correct classification ratio and the average portion of the search space that had to be searched is particularly interesting. Here we can see that when comparing each query points to only 2.4% of the base points on average, there is a roughly 83% chance of successfully finding the closest match.

Potential matches found	239775018
Comparisons per query vector	23977.5018
Average portion of search space searched	2.3978%
Correct classification ratio	83.03%

Table 5.1: Details (same for all optimization levels presented)

Performance when fully optimized:

	Time	**% of total time**
Total time	3.46s	100%
Calculating hash values for base vectors	0.726s	21.0%
Constructing LSH tables	0.247s	7.14%
Calculating hash values for query vectors	0.008s	0.231%
Finding potential matches	1.53s	44.2%
Matching potential matches	0.574s	16.6%

Table 5.2: Run-time for each phase after full optimization. Same table as Table 5.8, but also shown here underneath the information in Table 5.1 to increase readability of the final results.

Before moving on to the performance measurements of the algorithm I feel obliged to present information about how the time of the program is measured. Finding the time of each individual phase of the program is done by measuring time before and after the execution of that phase, the time is taken using functions from `<sys/time.h>`.

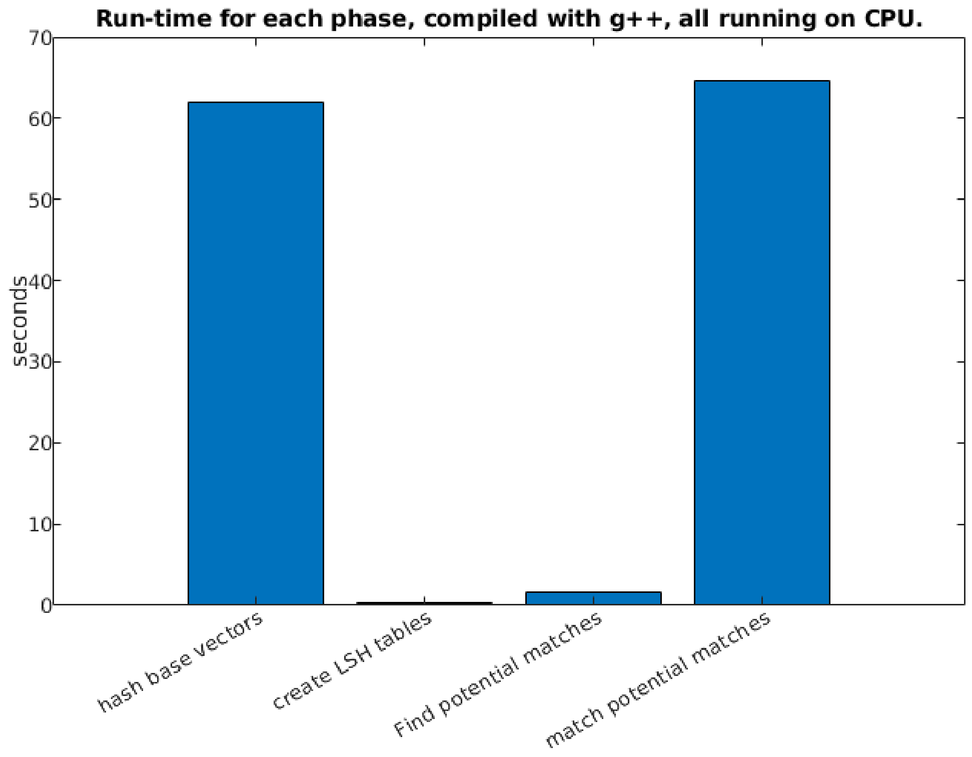
When taking the total time however, the `time` command in the bash shell was used instead. This leads to various unrelated part of the program being part of the total time presented (reading files, printing output, etc), and this is why the percentages of the different phases of the program don't necessarily add up to the total time. Still, the important parts of the program are really the performance of each of its major phases, and the most interesting aspects of the program in relation to OpenACC is how much each of these phases improve for each optimization stage, and how much time they take in relation to each other.

5.1.1 CPU program, g++ compiler

When running the program on the CPU using the g++ compiler we get the results in Table 5.3. We can see that the two most time consuming parts of the program are calculating hash values for base vectors, and matching the query points to their potential matches.

	Time	**% of total time**
Total time	129s	100%
Calculating hash values for base vectors	61.9s	48.0%
Constructing LSH tables	0.221s	0.171%
Calculating hash values for query vectors	0.583s	0.452%
Finding potential matches	1.54s	1.19%
Matching potential matches	64.6s	50.1%

Table 5.3: Run-time for each phase, compiled with g++, all running on CPU.



5.1.2 CPU program, pgc++ compiler

When changing from the g++ to the pgc++ compiler the time taken for the parallelizable part of the program decreases substantially. Although I am not adding any OpenACC directives and the program executes on a single CPU core, the pgc++ compiler optimizes the program using SIMD instructions.

We can see that the time required to calculate hash values for base vectors reduces from 61.9 seconds to 7.93 seconds resulting in a 7.81x speedup. Likewise, the amount of time required to find the best match among potential matches reduces from 64.6 seconds to 26.0 seconds resulting in a 2.48x speedup.

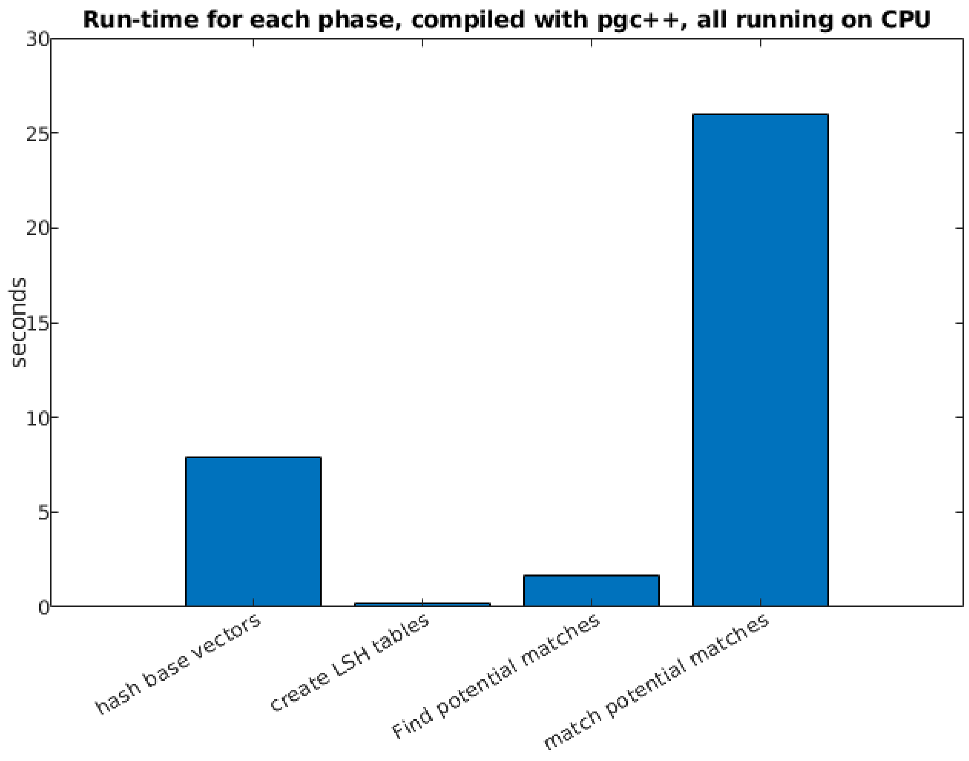
The execution time of the entire program went from 129 seconds to 36.2 seconds, which is a 3.56x speedup.

The reason why I introduced this step when measuring the results of the algorithm was because I wanted to compare the different optimization stages (when offloading work to the GPU) to the best single core CPU version that was easily available to me.

I also presented the results when using the g++ compiler because I think it might be interesting to see the differences in run-times when using the two compilers. It is interesting to see how much the implementation is accelerated by the pgc++ compiler using SIMD instructions.

	Time	% of total time
Total time	36.2s	100%
Calculating hash values for base vectors	7.93s	21.9%
Constructing LSH tables	0.234s	0.646%
Calculating hash values for query vectors	0.084s	0.232%
Finding potential matches	1.63s	4.50%
Matching potential matches	26.0s	71.8%

Table 5.4: Run-time for each phase, compiled with pgc++, all running on CPU.



5.1.3 Parallelized hashing

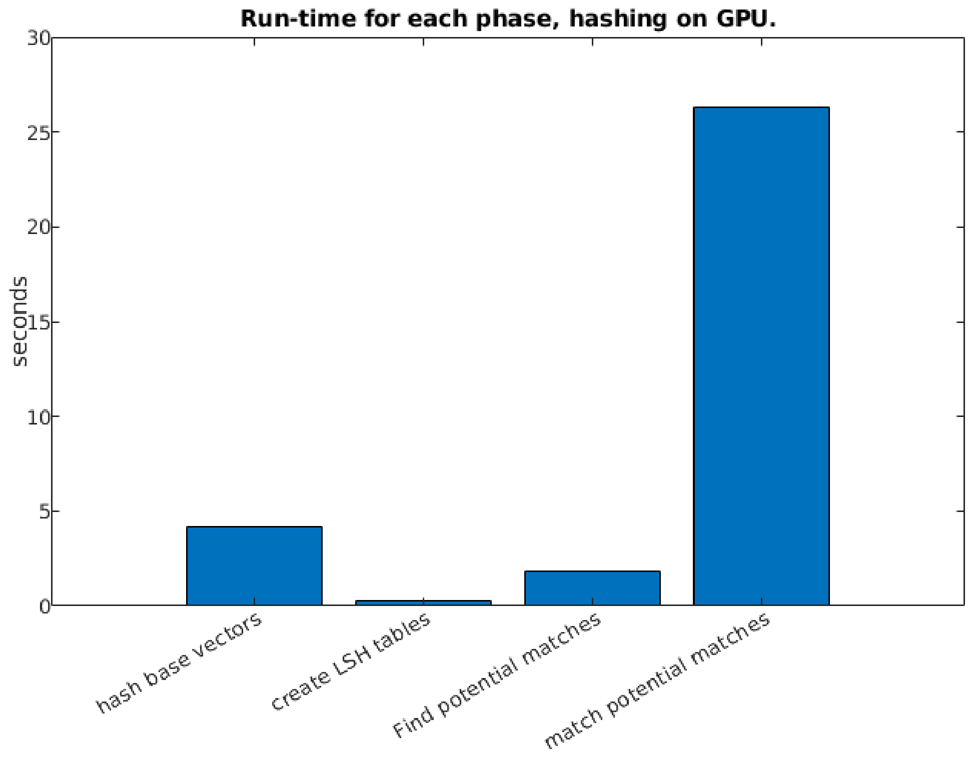
After moving computation of the hashing part of the function from the CPU to the GPU we end up with the run-times in Table 5.5.

As we can see the time spent on calculating hash values for base vectors went from 7.93 seconds to 4.19 seconds, which is a 1.89x (89%) speedup.

The time taken for the entire program remains largely the same however since the point matching remains the most time consuming part of the program. The total time spent went from 36.2 to 33.0 which is a 9.7% speedup. This speedup is still substantial however, since the matching of points which are currently the bottleneck of the program will see a massive speedup in the next optimization stage.

	Time	**% of total time**
Total time	33.0s	100%
Calculating hash values for base vectors	4.19s	12.7%
Constructing LSH tables	0.263s	0.797%
Calculating hash values for query vectors	0.054s	1.64%
Finding potential matches	1.83s	5.55%
Matching potential matches	26.3s	79.7%

Table 5.5: Run-time for each phase, hashing on GPU.



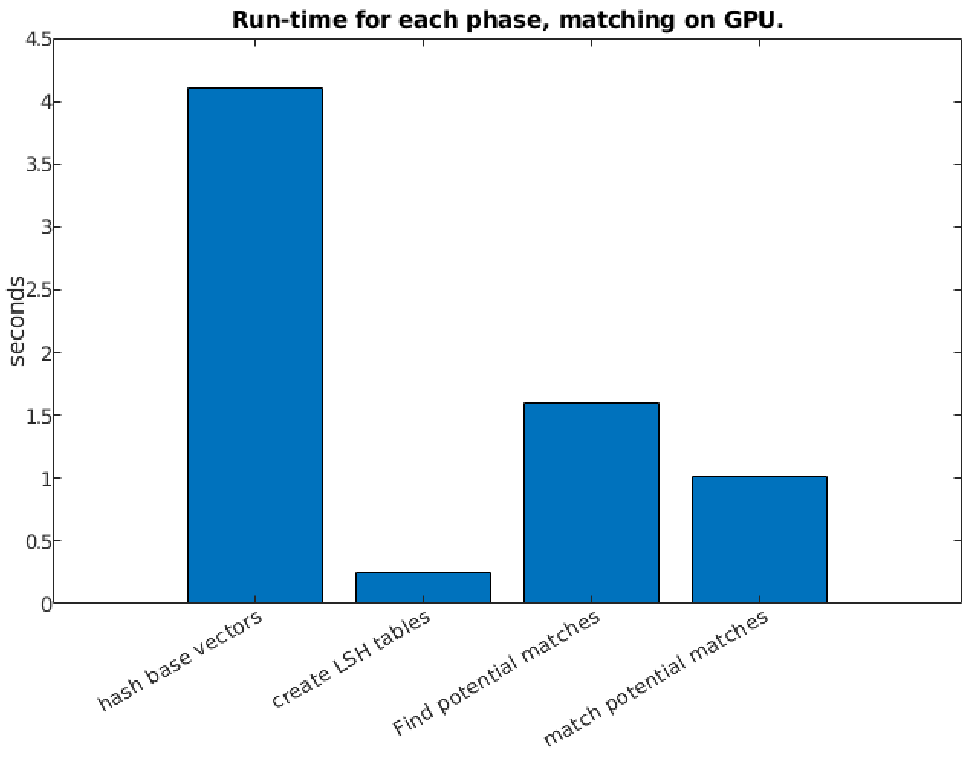
5.1.4 Parallelized matching

Moving computation of the point matching from the CPU to the GPU makes a massive difference on performance as one can see from the new run-times in Table 5.6.

The point matching phase of the program went from taking 26.3 seconds to taking 1.01 seconds resulting in a 26.4x speedup. This phase of the program went from taking up 79.7% of the time to only 24.6% of the time. This turns this phase of the program from taking over three quarters of the time to taking roughly a quarter of the time, which turns it from a major bottleneck to simply a large time consuming portion.

	Time	% of total time
Total time	7.39s	100%
Calculating hash values for base vectors	4.11s	55.6%
Constructing LSH tables	0.255s	6.20%
Calculating hash values for query vectors	0.055s	1.34%
Finding potential matches	1.60s	38.9%
Matching potential matches	1.01s	24.6%

Table 5.6: Run-time for each phase, matching on GPU.



5.1.5 Loop optimization

Table 5.7 shows the run-times of parts of the program after performing loop optimization for both the hashing and matching function.

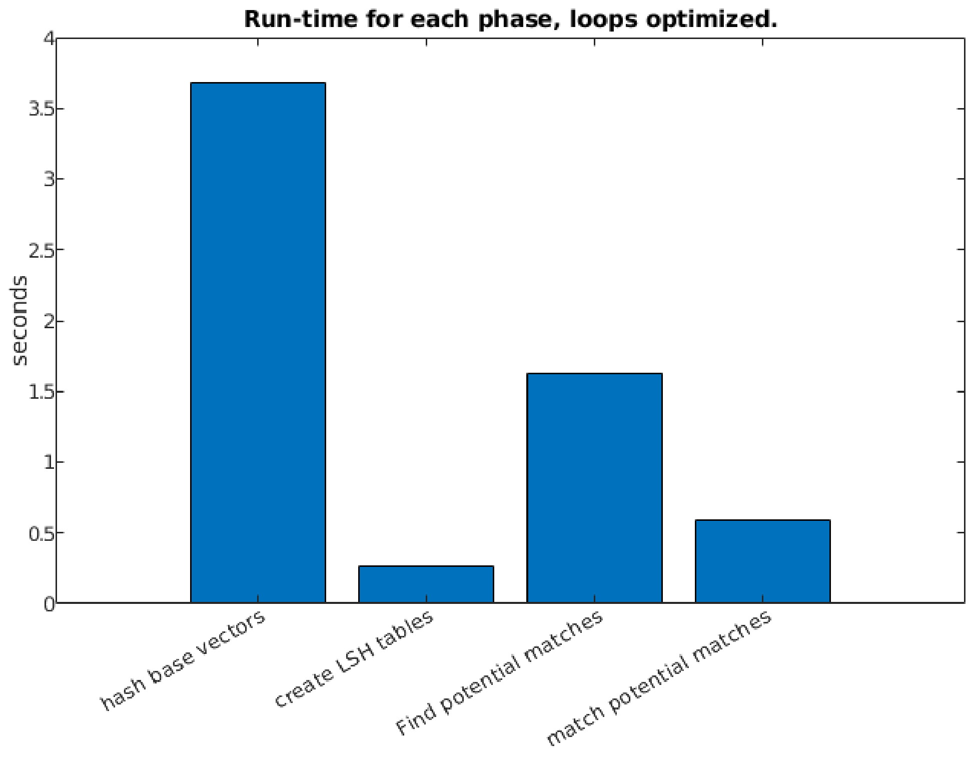
The use of loop optimizations leads to a small but sizable improvement in the time spent on calculating the hash values for base vectors. This part of the program went from taking 4.11 seconds to taking 3.68 seconds and received a 1.11x (11%) incremental speedup. This phase of the program received a 2.15x speedup from the CPU version compiled with pgc++, and a 16.8x speedup from the serial CPU version compiled with g++.

The point matching phase however previously took 1.01 seconds to taking 0.584 seconds, resulting in a significant 1.73x (73%) incremental speedup from the previous optimization stage. This phase of the program received a 44.5x speedup from the CPU version compiled with pgc++, and a 110.6x speedup from the serial CPU version compiled with g++.

The total time of the program improves from 7.39 seconds to 6.65 seconds, which is a 1.11x speedup (which happens to be the same amount of speedup achieved by the hashing phase).

	Time	**% of total time**
Total time	6.65s	100%
Calculating hash values for base vectors	3.68s	55.3%
Constructing LSH tables	0.269s	4.05%
Calculating hash values for query vectors	0.044s	0.616%
Finding potential matches	1.63s	24.5%
Matching potential matches	0.584s	8.78%

Table 5.7: Run-time for each phase, loops optimized.



5.1.6 Moving memory in bulk

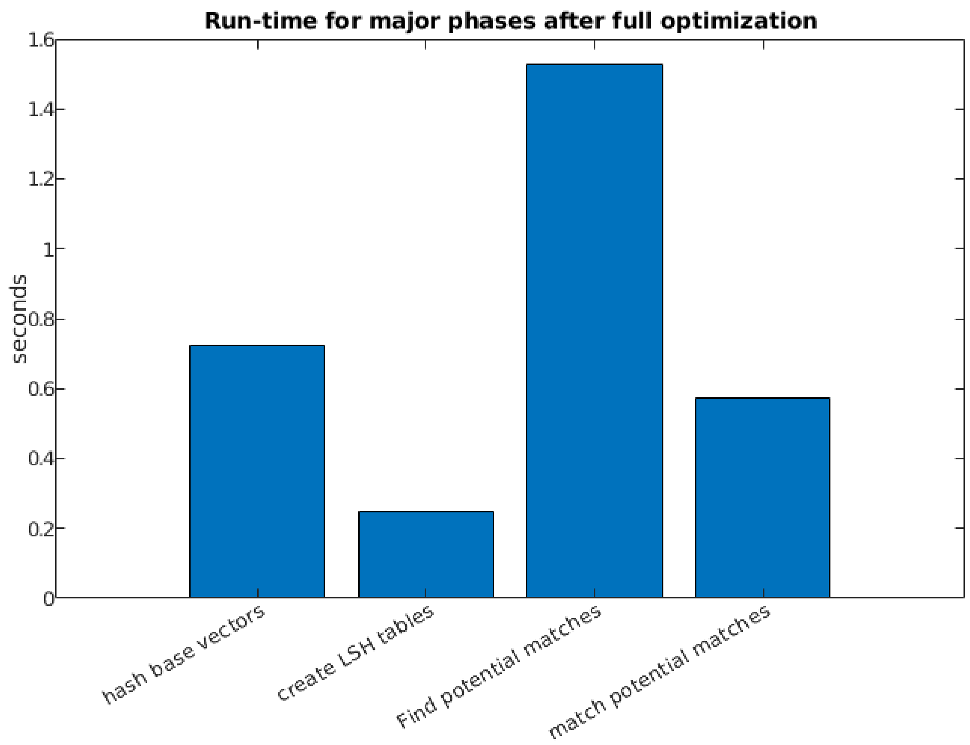
By optimizing memory movement some more for the hashing function we can see a substantial improvement in the run-time. The time spent on hashing the base vectors decreased from 3.68 seconds to 0.726 seconds, resulting in a 5.07x speedup from the previous optimization. This part of the program had a 10.9x total speedup from the CPU version of the program compiled with pgc++, and an 85.3x total speedup from the CPU version compiled with g++.

The time spent on the entire program is now 3.46 seconds (this includes IO and other overhead), for the CPU version it was 129 seconds. This means the entire program had a 10.5x speedup from the CPU version of the program compiled with pgc++. Or a 37.3x speedup from the CPU version compiled with g++.

One can see from Table 5.8 that the bottleneck of the program is no longer the raw vector multiplications associated with the hashing of the base vectors, nor the raw point comparisons associated with matching the query point with its potential matches. The most expensive part of the program is instead finding potential matches by looking up which base points a query point should be compared to in the LSH tables. This function is not optimized in this thesis.

	Time	**% of total time**
Total time	3.46s	100%
Calculating hash values for base vectors	0.726s	21.0%
Constructing LSH tables	0.247s	7.14%
Calculating hash values for query vectors	0.008s	0.231%
Finding potential matches	1.53s	44.2%
Matching potential matches	0.574s	16.6%

Table 5.8: Run-time for each phase, data locality optimized further.



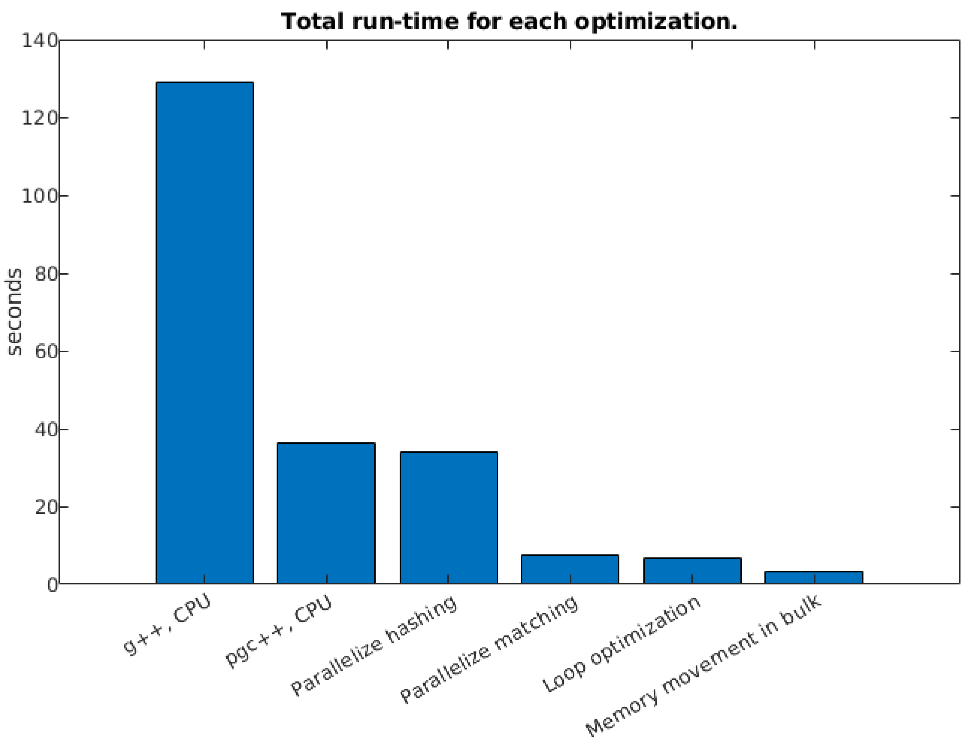
The speedup acquired through the use of OpenACC demonstrates that OpenACC was very effective in accelerating this algorithm. The reason for this is to a large extent designed to be easily parallelizable, if one wants to make a program parallel it is usually best to have that in mind from the start. In this case, much of the computation was based on operations on vectors, these types of operations are easy to parallelize.

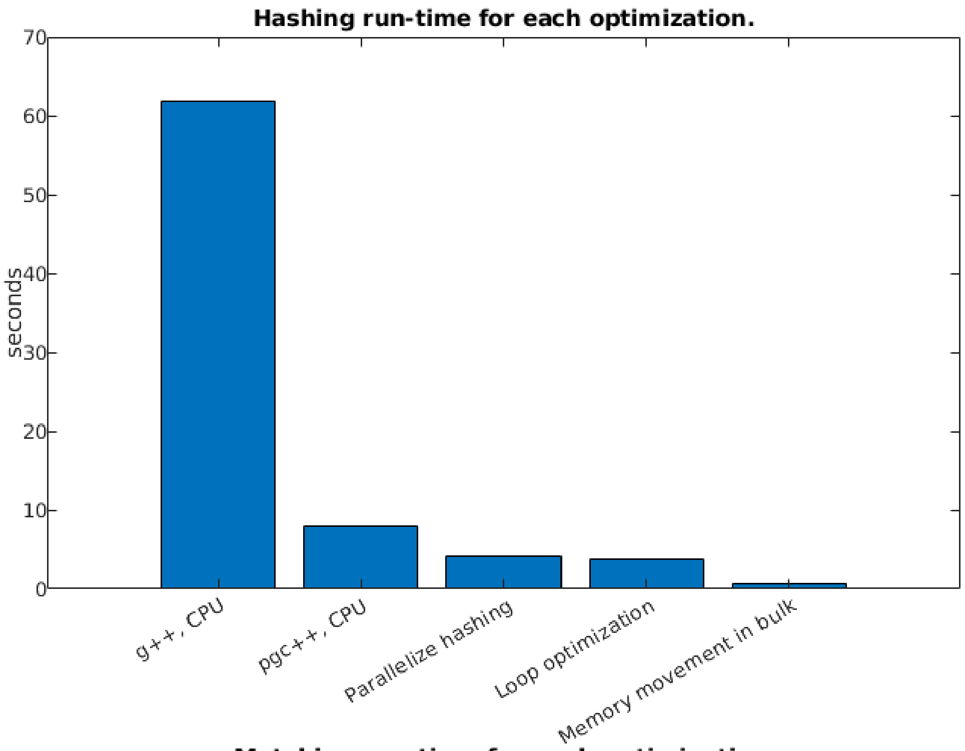
5.2 Performance of Each Optimization Summarized

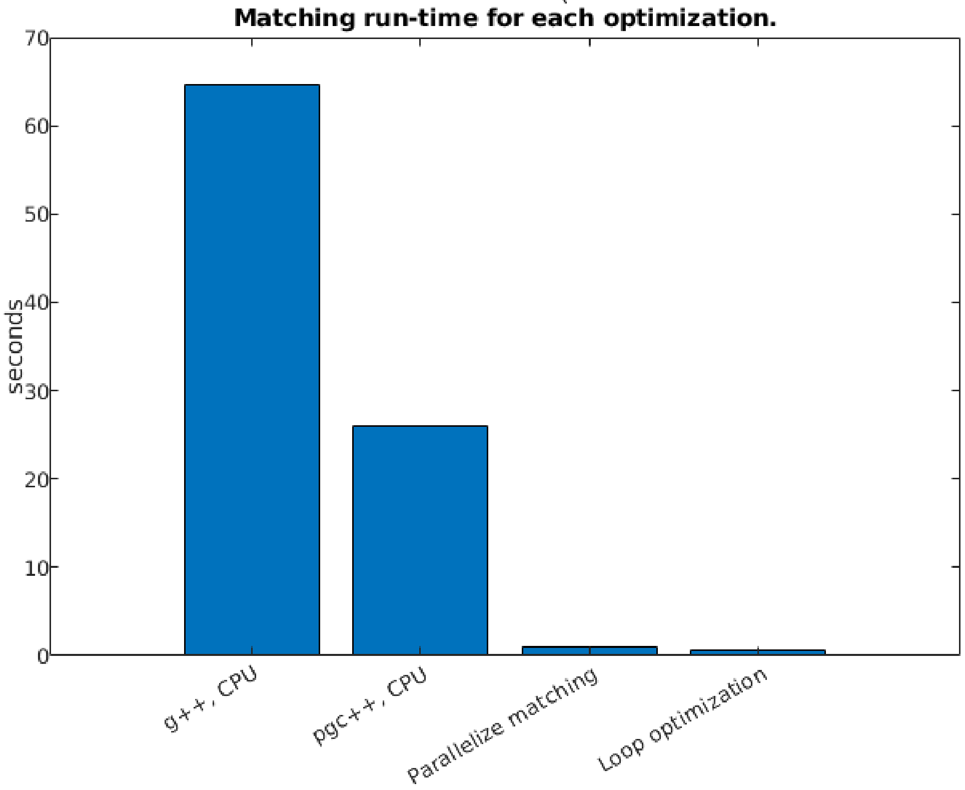
In this section I present some tables that summarize the timing and speedup for each stage of optimization.

	Total time	**Hashing**	**Finding matches**
CPU program, g++	129s	61.9s	64.6s
CPU program, pgc++	36.2s	7.93s	26.0s
Parallelize hashing	33.9s	4.19s	26.3s
Parallelize matching	7.39s	4.11s	1.01s
Loop optimization	6.65s	3.68	0.584
Moving memory in bulk	3.46s	0.726	0.574

Table 5.9: Run-time results




Hashing run-time for each optimization.
seconds
0
10
20
30
40
50
60
70
g++, CPU
pgc++, CPU
Parallelize hashing
Loop optimization
Memory movement in bulk


Matching run-time for each optimization.
seconds
0
10
20
30
40
50
60
70
g++, CPU
pgc++, CPU
Parallelize matching
Loop optimization

	Time	Speedup from CPU (g++)	Speedup from CPU (pgc++)	Incremental speedup
CPU, g++	61.9s	1x	-	-
CPU, pgc++	7.93s	7.81x	1x	7.81x
Parallelize	4.19s	14.8x	1.89x	1.89x
Loop optimization	3.68s	16.8x	2.15x	1.14x
Moving memory in bulk	0.726s	85.3x	10.9x	5.07x

Table 5.10: Hashing run-times and speed-ups

	Time	Speedup from CPU (g++)	Speedup from CPU (pgc++)	Incremental speedup
CPU, g++	64.6s	1x	-	-
CPU, pgc++	26.0s	2.48x	1x	2.48x
Parallelize	1.01s	64.0x	25.7x	25.7x
Loop optimization	0.584s	110.6x	44.5x	1.73x

Table 5.11: Matching run-times and speed-ups

5.3 Different Numbers of Hyperplanes

From Table 5.12 we can see that the number of correct matches goes down for each hyperplane added, this is because the chances of the query point being separated from its best match in the base set increases with every hyperplane splitting the search space.

The average portion of the search space searched also decreases for every hyperplane added since every hyperplane splits the search space further.

Likewise, in the beginning, when the portion of search space searched goes down, the time required by the algorithm goes down with it, this happens as long as the timing is dominated by the time spent on point matching.

We see from Table 5.14 that with around 20 to 21 hyperplanes, that the phases of calculating hash values and constructing LSH tables start to make up a majority of the time taken by the algorithm. This is because adding more hyperplanes will increase the overhead of the program, these two functions in particular will start taking up more and more time for each hyperplane added eventually becoming bottlenecks of the program as long as the number of base points remain the same.

Number of hyperplanes	Time taken	Correct ratio	Average % of search space searched
15	5.62s	87.1%	3.55%
16	4.20s	83.0%	2.40%
17	3.77s	79.6%	2.00%
18	3.11s	75.0%	1.30%
19	3.11s	71.8%	1.11%
20	3.00s	67.2%	0.770%
21	3.39s	62.9%	0.575%
22	4.45s	58.0%	0.443%

Table 5.12: Results with different numbers of hyperplanes and 32 tables

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH tables	Finding potential matches	Matching potential matches
15	0.739s	0.212s	3.001s	1.018s
16	0.746s	0.238s	1.954s	0.679s
17	0.831s	0.226s	1.570s	0.535s
18	0.854s	0.329s	1.006s	0.375s
19	0.838s	0.423s	0.904s	0.395s
20	0.901s	0.629s	0.669s	0.257s
21	0.867s	1.105s	0.601s	0.242s
22	0.925s	2.060s	0.636s	0.214s

Table 5.13: Time with different numbers of hyperplanes and 32 tables

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH tables	Finding potential matches	Matching potential matches
15	13.1%	3.77%	53.4%	18.1%
16	17.8%	5.66%	46.5%	16.1%
17	22.0%	5.99%	41.6%	14.2%
18	27.5%	10.6%	32.3%	12.1%
19	26.9%	13.6%	29.1%	12.7%
20	30.0%	21.0%	22.3%	8.57%
21	25.6%	32.6%	17.7%	7.14%
22	20.8%	46.3%	14.3%	4.81%

Table 5.14: Percentage of total time spent on each phase with different numbers of hyperplanes and 32 tables

5.4 Different Numbers of tables

The effects of increasing the number of tables are quite clear from looking at Table 5.15:

- More of the search search space is searched.
- The time taken by the algorithm increases since more of the search space is searched and every query point has to be hashed for each table.
- The correct ratio increases as a result of each query point being compared to even more nearby points

The time taken for each individual phase of the algorithm also increases due to the scale of operations increasing as we see in Table 5.16.

I should mention again that the total time is taken with the `time` command which includes some overhead, so the percentages in 5.17 does not add up to 100%. This makes comparing the evolution of percentages not ideal, and it should be kept in mind, particularly to explain what is happening for the portion of time spent on "matching potential matches" when more LSH tables are added.

We can see some different changes in the portion of time being spent in each part of the algorithm in Table 5.17. While the time spent on hashing increases for each table added it does so very slowly compared to the other functions. When going from 2 to 4 tables the amount of computation required for hashing the base points double, the time however only increases from 0.290 to 0.322 seconds, which is only a 9% increase, this is most likely due to a sort of "economics of scale" for GPUs, where time required to do more work

When seeing the combined time as being the time for all these phases, it uses 0.570 seconds with two LSH tables, and 3.599 seconds with 32 LSH tables. We can see from the total time taken with the time command that approximately half a second is used on different types of overhead not related to the program.

When using this "combined time" instead of the time from the `time` command, the portion of time spent on calculating hash values goes from 50.9% to 21.4%. For constructing LSH tables it goes from 2.46% to 6.72%. For finding potential matches it goes from 21.8% to 53.7%. For matching potential matches it goes from 26.5% to 18.1%.

Number of LSH tables	**Time taken**	**Correct ratio**	**Average % of search space searched**
2	1.06s	20.5%	0.220%
4	1.12s	31.1%	0.324%
6	1.38s	40.4%	0.460%
8	1.57s	48.1%	0.625%
10	1.86s	53.8%	0.777%
12	2.12s	59.5%	0.994%
14	2.39s	63.7%	1.15%
16	2.54s	67.0%	1.31%
18	2.79s	69.8%	1.43%
20	2.95s	72.2%	1.54%
22	3.20s	74.3%	1.68%
24	3.41s	76.8%	1.84%
26	3.57s	78.6%	1.98%
28	3.71s	79.6%	2.04%
30	3.94s	81.3%	2.22%
32	4.16s	83.0%	2.40%

Table 5.15: Results with 16 hyperplanes and different number of LSH tables

Number of LSH tables	**Calculating hash values for base vectors**	**Constructing LSH tables**	**Finding potential matches**	**Matching potential matches**
2	0.290s	0.014s	0.124s	0.151s
4	0.322s	0.030s	0.191s	0.172s
6	0.363s	0.045s	0.281s	0.199s
8	0.404s	0.061s	0.391s	0.216s
10	0.450s	0.075s	0.504s	0.320s
12	0.476s	0.089s	0.673s	0.359s
14	0.517s	0.104s	0.800s	0.433s
16	0.532s	0.121s	0.902s	0.457s
18	0.573s	0.135s	1.059s	0.501s
20	0.591s	0.150s	1.143s	0.540s
22	0.602s	0.168s	1.284s	0.616s
24	0.633s	0.179s	1.436s	0.626s
26	0.687s	0.193s	1.550s	0.597s
28	0.700s	0.210s	1.649s	0.593s
30	0.722s	0.227s	1.841s	0.600s
32	0.771s	0.242s	1.934s	0.652s

Table 5.16: Time for each part of the program with 16 hyperplanes and different number of LSH tables

Number of LSH tables	Calculating hash values for base vectors	Constructing LSH tables	Finding potential matches	Matching potential matches
2	27.4%	1.32%	11.7%	14.5%
4	28.8%	1.24%	11.1%	13.5%
6	26.3%	3.26%	20.4%	14.4%
8	25.7%	3.89%	24.9%	13.8%
10	24.2%	4.03%	27.1%	15.2%
12	22.5%	4.20%	31.7%	16.9%
14	21.6%	4.35%	33.5%	18.1%
16	20.9%	4.76%	35.5%	18.0%
18	20.5%	4.84%	38.0%	18.0%
20	20.0%	5.08%	38.7%	18.3%
22	18.8%	5.25%	40.1%	19.3%
24	18.6%	5.25%	42.1%	18.4%
26	19.2%	5.41%	43.4%	16.7%
28	18.9%	5.66%	44.4%	16.0%
30	18.3%	5.76%	46.7%	15.2%
32	18.5%	5.82%	46.5%	15.7%

Table 5.17: Percentage of total time for each part of the program with 16 hyperplanes and different number of LSH tables

5.5 Different Hyperplanes

Just to show that the results are stable using different sets of hyperplanes, I present the statistics shown in the tables in this section.

The hyperplanes are all random except for the fact that the vectors representing them are all of length one with elements that sum to zero.

Hyperplane set nr.	Time taken	Recal	Average % of search space searched
1	4.31s	83.0%	2.40%
2	4.16s	83.1%	2.29%
3	3.96s	82.9%	2.27%
4	4.55s	85.2%	2.81%
5	3.93s	83.4%	2.29%
6	3.97s	83.6%	2.31%

Table 5.18: Results with 16 hyperplanes and 32 tables, the hyperplanes are represented by vectors that are of length 1 with elements that sum to zero, but are otherwise completely random. Each row represents different sets of random hyperplanes.

Hyperplane set nr.	Calculating hash values for base vectors	Constructing LSH tables	Finding potential matches	Matching potential matches
1	0.776s	0.244s	1.925s	0.778s
2	0.801s	0.241s	1.814s	0.710s
3	0.756s	0.241s	1.791s	0.602s
4	0.753s	0.235s	2.245s	0.757s
5	0.770s	0.239s	1.729s	0.645s
6	0.756s	0.238s	1.793s	0.628s

Table 5.19: Time with 16 hyperplanes and 32 tables, the hyperplanes are represented by vectors that are of length 1 with elements that sum to zero, but are otherwise completely random. Each row represents different sets of random hyperplanes.

Hyperplane set nr.	Calculating hash values for base vectors	Constructing LSH tables	Finding potential matches	Matching potential matches
1	18.0%	5.66%	44.7%	18.1%
2	19.3%	5.79%	43.6%	17.1%
3	19.1%	60.9%	45.2%	15.2%
4	16.5%	5.16%	49.3%	16.6%
5	19.6%	6.08%	44.0%	16.4%
6	19.4%	5.99%	45.2%	15.8%

Table 5.20: Percentage of total time spent on each phase with 16 hyperplanes and 32 tables, the hyperplanes are represented by vectors that are of length 1 with elements that sum to zero, but are otherwise completely random. Each row represents different sets of random hyperplanes.

5.6 Small Dataset Results

For the most part this thesis examines results for a large dataset from the paper called "Product Quantization for Nearest Neighbor Search"[7] with one million base points and ten thousand query points. In this short section however, I will examine some results for a smaller dataset associated with the same paper. This dataset contains 10 000 base points and 100 query vectors.

For the smaller dataset, the results are less stable since time measurements for smaller programs tend to be more volatile in general. Additionally, both the LSH algorithm and GPU computing are aimed at optimizing large scale problems, so for very small datasets a linear CPU solution might do the trick better than locality sensitive hashing optimized with GPU computing. Still, it might be interesting to highlight details about the algorithm when working on a smaller dataset, and contrast these results with those of one gets when applying the algorithm to a larger dataset.

This section only contains tables and no plots, this is because the results for the small dataset are not very important compared to the results for the large datasets. Plots and graphs are therefore not presented in order to limit the space of these sections and not allow them to detract from the more important sections.

When using a smaller dataset it is generally a good idea to also reduce the number of hyperplanes used per table. First I will show results for 16 hyperplanes (which is the same number of hyperplanes as what was used for the large dataset), then I will show the results for 8 hyperplanes. Throughout this section, 32 tables will be used. For 16 hyperplanes we get the results:

Potential matches found	26700
Comparisons per query vector	267
Average portion of search space searched	2.67%
Correct classification ratio	81.0%

Table 5.21: Statistics about the program. Run on the small dataset. 16 hyperplanes. 32 tables. The details are the same for all optimization levels presented.

	Time	**% of total time**
Total time	0.618s	100%
Calculating hash values for base vectors	0.577s	93.4%
Constructing LSH tables	0.018s	2.91%
Calculating hash values for query vectors	0.006s	0.971%
Finding potential matches	0.001s	0.162%
Matching potential matches	0.004s	0.647%

Table 5.22: Run-time for each phase, compiled with g++, all running on CPU. Number of hyperplanes = 16, number of tables = 32.

	Time	**% of total time**
Total time	0.110s	100%
Calculating hash values for base vectors	0.074s	67.3%
Constructing LSH tables	0.018s	16.4%
Calculating hash values for query vectors	0.001s	0.909%
Finding potential matches	0.001s	0.909%
Matching potential matches	0.001s	0.909%

Table 5.23: Run-time for each phase, compiled with pgc++ (using SIMD parallelization), all running on CPU. Number of hyperplanes = 16, number of tables = 32.

	Time	**% of total time**
Total time	0.265s	100%
Calculating hash values for base vectors	0.184s	69.4%
Constructing LSH tables	0.021s	7.92%
Calculating hash values for query vectors	0.003s	1.13%
Finding potential matches	0.000s	$\sim$0%
Matching potential matches	0.003s	1.13%

Table 5.24: Run-time when "Calculating hash values" and "Matching potential matches" are run on GPUs and the program is fully optimized. Number of hyperplanes = 16, number of tables = 32.

For the large dataset we were able to get an 83.0% correct ratio while only searching approximately 2.40% of the search space. For the smaller dataset we get an 81.0% while searching 2.67% of the search space. The correct ratio is here smaller compared to the amount of search space searched, the results are still similar enough to be comparable with each other however.

We can see that since there are only 10 000 base points, and we are using 32 tables with 16 hyperplanes each, the algorithm spends the vast majority of its time hashing values. The number of hyperplanes (and maybe also the number of tables) are thus way too large for this problem size.

We can also see that for this small dataset, the CPU version compiled with the `pgc++` compiler is the fastest of the three approaches, this is because it accelerates the algorithm using SIMD operations making it faster than the serial version. For smaller problems such as this, CPU SIMD operations are generally faster than moving computation over to the GPU.

We do see however, that the GPU implementation outperforms the CPU implementation when it is not allowed to use SIMD operations. Even on this small problem scale therefore, the GPU implementation outperforms the pure CPU implementation, though this is mostly due to the relatively high number of tables used which scales up the amount of computation needed.

Most of the time spent on computation by the GPU is really spent on various types of overhead associated with running programs on GPUs such as moving memory. This overhead factors into the calculation of hash values since that is the first program that uses the GPU. When the matching phase starts, a lot of the memory that is needed is already located on the GPU, much of the other overhead related to initialization has also already taken place.

When using 8 hyperplanes:

Potential matches found	299127
Comparisons per query vector	2991.27
Average portion of search space searched	29.9%
Correct classification ratio	99.0%

Table 5.25: Statistics about the program. Run on the small dataset. 8 hyperplanes. 32 tables. The details are the same for all optimization levels presented.

	Time	**% of total time**
Total time	0.350s	100%
Calculating hash values for base vectors	0.290s	82.9%
Constructing LSH tables	0.001s	0.286%
Calculating hash values for query vectors	0.003s	0.857%
Finding potential matches	0.004s	1.14%
Matching potential matches	0.042s	12.0%

Table 5.26: Run-time for each phase, compiled with g++, all running on CPU. Number of hyperplanes = 8, number of tables = 32.

	Time	**% of total time**
Total time	0.065s	100%
Calculating hash values for base vectors	0.036s	55.4%
Constructing LSH tables	0.001s	1.54%
Calculating hash values for query vectors	0.001s	1.54%
Finding potential matches	0.003s	4.61%
Matching potential matches	0.011s	16.9%

Table 5.27: Run-time for each phase, compiled with pgc++ (using SIMD parallelization), all running on CPU. Number of hyperplanes = 8, number of tables = 32.

	Time	**% of total time**
Total time	0.229s	100%
Calculating hash values for base vectors	0.162s	70.7%
Constructing LSH tables	0.001s	0.437%
Calculating hash values for query vectors	0.002s	0.873%
Finding potential matches	0.003s	1.31%
Matching potential matches	0.006s	2.62%

Table 5.28: Run-time when "Calculating hash values" and "Matching potential matches" are run on GPUs and the program is fully optimized. Number of hyperplanes = 8, number of tables = 32.

When using only 8 hyperplanes, the correct classification ratio increases from 81.0% to 99.0%, meanwhile the average portion of the search space that is searched increases rapidly from 2.67% to 29.9%. At this point, almost a third of the search space is searched, and yet, the vast majority of the time is always spent on calculating hash values for the base vectors, which indicates that the problem is way too small in scale for locality sensitive hashing to be effective (at least with such a large number of tables, it may perform better if the number of tables is reduced).

For the versions of the program that ran on the CPU we can see that the time spent on hashing went significantly down from when 16 tables were used. In fact, the time spent halved, which makes sense since we use exactly half the amount of hyperplanes. Meanwhile the time spent matching potential matches went up by roughly a factor of 10, which also makes sense since slightly over ten times the number of potential matches had to be searched. It is clear however, that the time went down significantly altogether (by almost a factor of 2) since calculating hash values takes up the vast majority of time.

For the GPU implementation however, not much changed. The time spent on hashing went down marginally, whereas the time spent on matching doubled, however this is not very impactful on the performance of the algorithm overall. The vast majority of the time that falls under calculating hash values is overhead associated with OpenACC and not spent on actual computation. The GPU implementation does in this case still outperform the strictly serial version due to the scale of computation involved in the hashing phase, due to the number of tables used.

Chapter 6

Single Table LSH

6.1 Introduction

In the algorithm for LSH that I presented earlier, several LSH tables had to be used in order to increase the probability of a correct match, i.e. this was done to make up for the possibility that the query point and its best match happens to get hashed into different boxes due to them falling on opposite sides of a hyperplane in the LSH table.

Using several tables comes with a lot of extra resource requirements and computation time, each table needs to be represented separately, and one has to look for potential matches in all of them.

In Section 3.3.4, I discussed how one can divide the search space using hyperplanes; within this explanation I also pointed out how one can calculate not only which side of a hyperplane a point falls on, but also the distance from the point to that hyperplane.

The time when finding the distance between a point and the hyperplane might be useful is when one creates an implementation of LSH that checks the other side of a hyperplane if the distance between the point and that particular hyperplane is sufficiently low. When one finds the distance between the point and the hyperplane one can set a maximum distance allowed before checking the other side of a hyperplane, then compare the distances to this threshold.

6.2 One Hash Bit Allowed To Change At A Time

One can check the other side of a combination of several hyperplanes, but let us for simplicity discuss in this section what happens when one allows the algorithm to check on the other side of only one hyperplane at a time, i.e. only one bit in the hashcode is allowed to change at a time.

When only allowing one bit deviation from the hashcode this will limit the accuracy of LSH using a single table, there will therefore still be a need for multiple tables in order to get an acceptable accuracy when finding matches for SIFT feature descriptors.

In this case there will also be an upper limit on the number of boxes that can possibly be checked by a single query point which will be equal to one plus the number of hyperplanes, such an upper bound is handy when implementing the algorithm. As I previously pointed out however, this restriction comes at the cost that when using this implementation several tables are still needed, though perhaps not as many as before.

This method of checking several boxes per table increases the accuracy at the cost of speed without having to add even more tables. One can therefore get comparable results as with the original LSH implementation using substantially fewer tables, and thus less memory. Creating this version of the algorithm should also be straight forward once one has a working implementation of LSH.

The creation phase for this version of LSH is the same as before, however the searching phase will be a bit more complicated.

When calculating a hash value for a query point the algorithm must store, not only the hash codes themselves, but also either the distances between the query point and the hyperplanes, or some other piece of information (in the form of bit-masks or arrays of bytes) that indicates which hyperplanes came too close to the query point.

The program will then have to construct the hash code for the query point and the hash code for neighboring boxes that are sufficiently close. The hash codes for the neighboring boxes can be found using the xor binary operation and bit-masks.

Finally, to find potential matches in the LSH table for a given query point, the algorithm will have to search each group corresponding to all the hash codes in the list for that query point; this also has to be repeated for each table.

While all this sounds complicated, as long as one only allows a single bit of the hash code to deviate from its original value then the algorithm is only a small change from the original LSH algorithm.

6.3 Guaranteeing A Match Within Threshold

An interesting train of thought begins when one thinks about the possibility of checking all boxes within a given distance from the point, these boxes may be separated from query point's box by several hyperplanes, i.e. the hash code for these boxes may differ by several bits from the hash code of the query points.

One way to implement this might be to mark bits corresponding to hyperplanes that are too close to the query point, then check all combinations of these bits. For a reasonably high threshold however, this can in many cases quickly lead to too many boxes being checked with devastating effects on performance (in the case where all hyperplanes are too close for example, all the boxes will have to be checked, which would turn the algorithm into just a very ineffective linear search).

Instead of checking all possible combinations of bits corresponding to particularly nearby hyperplanes, one can instead try to add the distances for each hyperplane in order to potentially find the distance from the query point to the nearest part of the intersection between the hyperplanes in question. For random hyperplanes this is hard to do, but if one uses orthogonal hyperplanes instead then one can find the combination of distances using an extended version of the Pythagorean Theorem.

On Figure 6.1 I describe how one can check if the intersection of two hyperplanes are below a certain threshold using the distances from orthogonal hyperplanes and adding them together using the Pythagorean Theorem. Only two hyperplanes are used/shown, however one can use almost the same formula to find the distance to the intersection of any number of hyperplanes, as shown in Equation 6.1.

$$d_{1,2,\ldots,N}^2 = d_1^2 + d_2^2 + \ldots + d_N^2 \tag{6.1}$$

Using this formula one can find the minimum distance of any combination of orthogonal hyperplanes. This distance can then be compared to a threshold, and if the distance is lower than the threshold then the algorithm will decide to check on the other side of that combination of hyperplanes as well.

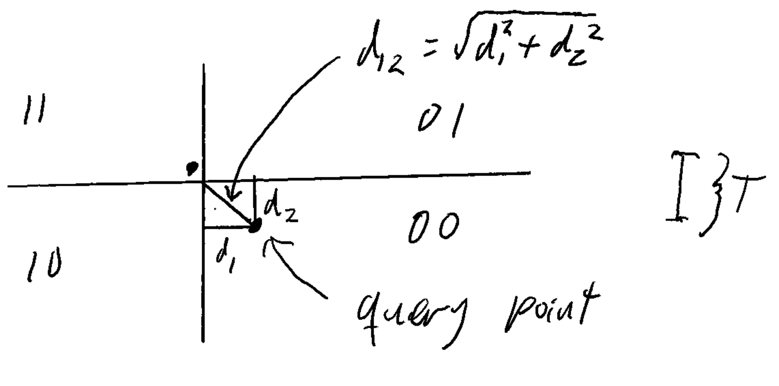
The hash code for the box in question is found by setting the bits corresponding to each hyperplane in the combination of hyperplanes to one in a mask where the other bits are zero, then xor'ing that result with the original hash code.

The naive way of checking nearby boxes would be to repeat the procedure outlined above for all possible combinations of hyperplanes. This would understandably require a huge amount of time, and will therefore not be practical in any real application since this will make the algorithm's overhead alone possibly rival that of naive matching (depending on how many hyperplanes are used). Instead of naively checking all possible combinations of hyperplanes, there are more efficient methods, one of which I will cover in Section 6.5.

In theory, the threshold one sets for checking on the other side of a combination of hyperplanes really represents a guarantee where if the distance between the query point and its best match is below this threshold, it will be found. I say in theory, since this is not always true likely due to imperfect floating point operations and similar phenomena.

One would think that one can find the maximum distance of any of the query points and each of their best matches, then use that maximum distance as a threshold and thus find all the matches of the query points, at least in theory. For the problem at hand however, matching SIFT feature descriptors, this is not wise.

Due to the fact that hyperplanes in 128 dimensions are many-dimensional objects with large border regions, whereas feature descriptors are only 128 dimensional points in space, the distance from a feature descriptor to a hyperplane is typically much lower that its distance to another feature descriptor, even its best match. This means that even a threshold equal to the average distance between a query point and its best match would typically be much too high, leading to distances to almost all hyperplanes being significantly lower than the threshold; this would lead to huge amounts of boxes being checked and it would be very bad for performance.



Check 00

if $d_1^2 < T^2$, check 10

if $d_2^2 < T^2$, check 01

if $d_{12}^2 < T^2$, check 11

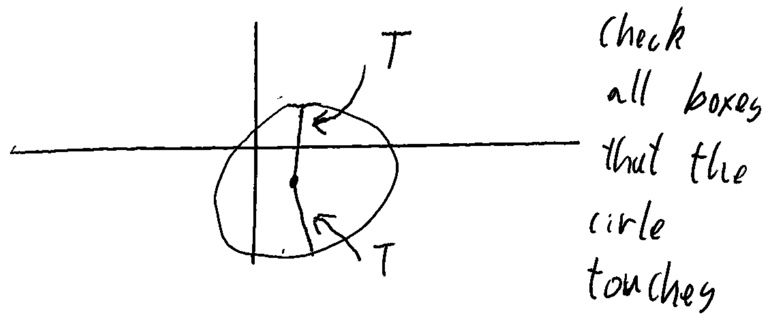


Figure 6.1: Figure showing how to check all possible boxes within a threshold T. The approximate size of the threshold is indicated in the figure on the top right. In this example, all boxes shown are within the threshold, thus they are all checked. Which boxes are checked can be visualized by drawing a circle with radius T around the query point as shown at the bottom of the figure, all the boxes that the circle touches has to be checked.

The good news is that, even without using a very large threshold to guarantee finding a best match, when using a smaller threshold the algorithm still gets good results, with accuracy and run-times that are comparable to those where several tables were used.

6.4 Alternative: Replicating Base Points Across Boxes

Before moving on to explain how one can find combined distances lower than the threshold in an efficient manner, it is also worth pointing out that there is another possible implementation that achieves the same goal as the one covered in the last section. The algorithm is very similar, but it does things in a different order and it comes with different performance trade-offs.

In Section 6.3 it was stated that one can check all boxes within a certain threshold of the query point in order to theoretically guarantee that points within that distance will be found; the logic being that one can combat the bad case where a query point falls particularly close to a hyperplane by checking on the other side of that hyperplane in these unfortunate cases. In this algorithm, the issue of points falling close to a hyperplane was solved by checking multiple boxes whose borders were sufficiently near the query point during the search phase; another way of achieving similar results would be to replicate references for base points to nearby boxes during the LSH table creation phase if they are sufficiently close to individual hyperplanes or certain combinations of hyperplanes.

Figure 6.2 attempts to show how this version of the algorithm would work. The upper part of the figure illustrates the version of the algorithm from the last section, where the algorithm checks distances from the query point to combinations of hyperplanes during the search phase, then checks neighboring boxes if the corresponding distances fall under a predefined threshold.

The bottom part of Figure 6.2 illustrates how the algorithm works for this alternative, second version. In this version, the algorithm takes the distance from each base point to each combination of hyperplanes, then if the distance is below the set threshold, the algorithm replicates the base point reference, in the LSH table, from its own box to a neighboring box at the other side of that combination of hyperplanes.

This can potentially provide a huge advantage during the search phase of the LSH algorithm. The algorithm would, in the search phase, only have to calculate a single hash value for each query point, and check only a single box (in the single LSH table) for potential matches, this would therefore make point look-ups faster compared to the case where multiple boxes would have to be checked for each point look-up in the search phase.

In this version of the algorithm we are replicating base points during the LSH table creation phase instead of searching multiple boxes for each query point during the matching phase, following this course of action will present different trade-offs than the implementation considered previously. The most obvious trade-off would be that we move complexity from the

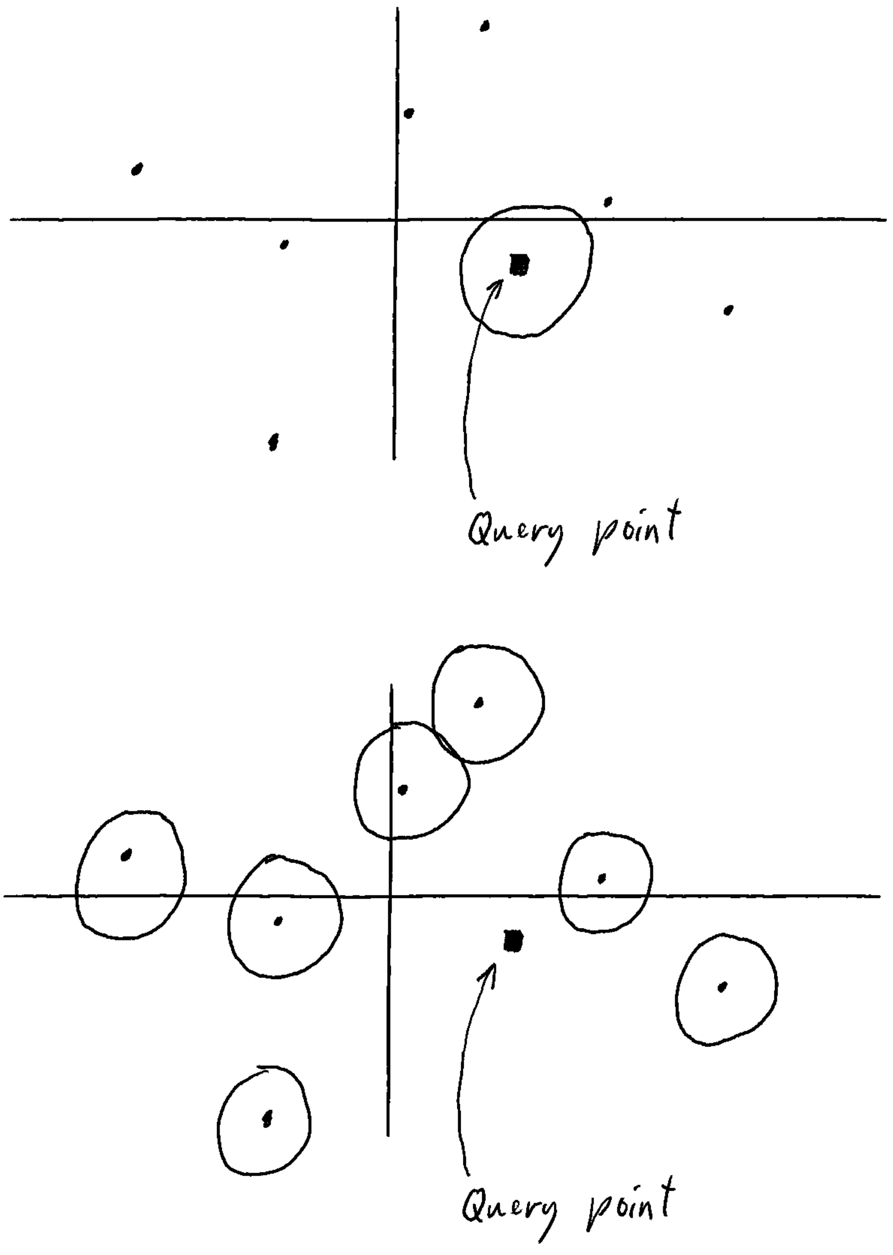


Figure 6.2: Figure showing on the bottom which points are copied over across the hyperplane to neighboring boxes. The circles should all be of the same size, and the radius represents the threshold for the maximum allowable distance to the hyperplanes. The algorithm will take the distances from each base point to each hyperplane and combinations of hyperplanes, then compare them to the threshold. The algorithm copies a base point over to the neighboring box if the distance to the combination of hyperplanes is below the threshold. One can see that the closest match to the query point in this case would have been be copied over the neighboring hyperplane to the same box as the query point is in.

search phase to the table creation phase, therefore the search phase will have the potential to be much quicker whereas the table creation phase will likely be much slower.

A more important disadvantage with this method however is that the amount of memory needed to store the LSH table becomes much larger when the base point references are replicated, since these references for each point will have to be replicated for each combination of hyperplanes with a combined distance lower than the threshold, this means that each point will have very many point references instead of just one and the size of the LSH table will increase correspondingly. An example: for cases where base points on average are closer than the threshold to 1000 different combinations of hyperplanes, the size of the LSH table would increase by a factor of 1000.

The number of correct matches will probably not be exactly the same for the two versions of the functions even when the same hyperplanes are used. A query point that is close to a hyperplane would be compared to every point in the neighboring box in the first version of the algorithm, in the second version however it would only be compared to the ones on the same border as they would be replicated across, it would also have to be compared to points that was copied over from other border regions even if the query point was not particularly close to that border region itself. Still, even if the number of correct matches will not be exactly equal for the two matches, they should be comparable when using the same input, it's just that both methods will sometimes find matches that the other one doesn't.

This version of the algorithm is presented since it offers very clear distinctive advantages, such as the fact that all suggested matches for a query point during the search phase can be found using only a single table lookup. This makes the algorithm ideal for cases where a single base set is searched a very large number of times (when the total number of query points for all query sets used to match this specific base set are much larger than the number of base points), i.e. the case where time spent creating the LSH table is dwarfed by the total amount of time spent searching these tables. The time spent creating the LSH table is probably not the greatest disadvantage with this algorithm however, that would be the amount of memory it requires.

To what extent the increased memory requirement would be an issue depends on the particularities of the application. If the query points are generally geographically close to their best match in the base set, this allows for setting a lower threshold for copying base points and thus makes the memory requirements much lower, increasing the viability of this version of the algorithm (though it must be said that query points being geographically close to their best matches in the base set increases the viability of all versions of LSH). If the amount of memory used increases to such an extent that one has to switch from storing the LSH table in primary memory to storing it in secondary memory, then that is a typical thing that can make this algorithm less effective.

When matching SIFT feature descriptors, the query points tend to be relatively far away from their best matches in relation to how far away they are from the hyperplanes (the distance from hyperplanes tend to be small due to the hyperplanes being many-dimensional objects making their boundaries very large). The threshold for copying points across boxes would therefore have to be quite high, meaning the algorithm will have to use very large amounts of memory. When testing an early implementation of this algorithm I found that it worked fine on a smaller dataset with 10 000 SIFT descriptors in the base set, however the algorithm quickly ran out of memory when working on 1 000 000 SIFT descriptors in the base set. In my tests I am not storing the LSH tables in secondary memory, therefore this version of the algorithm was not viable for my use case. I will therefore rather run and present results for the previous algorithm, which is effective both for the smaller and larger dataset.

6.5 Finding Accumulated Distances

The naive way of finding all possible combinations of hyperplanes with an accumulated squared distance less than a threshold[1] would be to try every possible combination and save the combinations where this is the case. As discussed previously however, while this approach is possible, it will take an impractically large amount of time. Instead, in this section, I aim to provide a more efficient alternative for doing this.

What I am essentially trying to do with this algorithm is combining subsets of hyperplanes where their accumulated squared distance values are lower than the threshold, then building a list of these sets such that all sets with an accumulated squared distance below the threshold are in that list.

First, I will describe how these sets of hyperplanes are represented. The sets are represented by bit masks where each bit represents a hyperplane, hyperplanes that are in the set are represented by '1' bits, and hyperplanes that are not in the set are represented by '0' bits. The combined distance for a given set is represented by a single floating point number. An array of bit masks (stored as integers) can therefore represent any collection of hyperplane sets, and an array of floating point numbers are used to represent the combined distances for every set in the list. It is important to note that, when constructing these sets, the order in which the hyperplanes are added to the sets will not matter, if one combines the set {1,2} with {3,4} to make the set {1,2,3,4}, that will be the same set as when one combines {1,3} and {2,4}; there is no order of hyperplanes included, this allows the algorithm to save a lot of work.

Back to the question of how to find all combinations of hyperplanes with a combined distance under the threshold. One way to try to solve this problem would be to maintain a sorted array of sets, kick out the

[1]The threshold here would be the square of the maximum distance allowed from a point to a hyperplane. This is the case since the threshold is compared to accumulated squared distance values from the point to each hyperplane that, when accumulated, represent the distance from the point to the intersection between these specific hyperplanes (using the Pythagorean formula).

elements with a squared distance above the threshold, then try to combine elements in the set in some intelligent way keeping the array sorted. This might provide some way of prioritizing finding the shortest distance sets first in the case where one might want to cut the process short. However, keeping a sorted algorithm requires a lot of extra overhead, so I therefore decided against it since I figured it would not be effective enough (and such a scheme seemed hard to work out).

Instead I decided to implement an algorithm that finds all combinations under the threshold while letting the array stay unsorted throughout the process. The algorithm starts by creating masks for every individual hyperplane with a squared distance lower than the threshold, then adding all those masks/subsets to a set/array. The algorithm then combines the subsets iteratively using two for-loops and pushes them to the end of the array.

In the outer loop, the algorithm iterates through the array starting at the second to last element and moving backwards towards the first element. In the inner loop, the algorithm will then attempt to combine the set at the current index of the loop with the sets located at all indices that come after it, i.e. combine the current set with all sets after it in the list. If the accumulated squared distance is lower than the threshold, a new set is created from the current set combined with the set it was compared with, then the new set is added to the end of the list.

Each hyperplane will then have attempted to combine with all the hyperplanes in the original list that comes after it, and also all combinations of the hyperplanes after it in the list that together have an accumulated squared distance below the threshold. This should ensure that by the end all combinations of hyperplanes that together have an accumulated squared distance below the threshold should have been added to the list.

The algorithm is perhaps best understood when looking at the code. I will present the code how it was implemented both in matlab and C so it can be best understood. The matlab code hopefully functions a bit like pseudocode in this case, and contributes to the understanding of this important algorithm:

```
function [setLen, combMasks, combDists] = ...
    findNeighborMasks(sqrdDists, threshold, nPlanes)

    % Masks and dists for combination of planes
    % (the lengths can be much larger than 20,
    %   20 is just the initial capacity)
    combMasks = cell(1,20);
    combDists = cell(1,20);

    % Add single values below the threshold to lists
    setLen = 0;
    for i = 1:nPlanes
        if (sqrdDists(i) < threshold)
            setLen = setLen + 1;
            combMasks{setLen} = bitshift(1,i-1);
            combDists{setLen} = sqrdDists(i);
        end
    end

    % For every value starting with the last, try to combine
    % the value with all other values in the list
    underThreshLen = setLen;
    for k = (underThreshLen-1):-1:1
        tmp = setLen;
        for i = (k+1):setLen
            if (combDists{k} < threshold)
                dist = combDists{k} + combDists{i};
                if (dist < threshold)
                    tmp = tmp + 1;
                    combMasks{tmp} =
                        bitor(combMasks{k}, combMasks{i});
                    combDists{tmp} = dist;
                end
            end
        end
        setLen = tmp;
    end
end
```

The code below is how the algorithm is implemented in C.

```
/**
 * Takes squared distances as arguments and finds all
 * non-ordered combinations of those distances that are
 * still lower than the threshold. Returns the masks
 * representing the combinations in the combMasks array, and
 * the accumulated distances in the combDists array.
 *
 * Have to reserve nGroups elements for both combMasks and
 * combDists for the function to be completely secure.
 * nGroups is the maximum amount of boxes that a point can
 * be mapped to.
 *
 * @param sqrdDists : The squared dist for each hyperplane
 *
 * @param nPlanes : The number of hyperplanes
 *                  (length of sqrdDists)
 *
 * @param threshold : The maximum distance allowed squared
 *
 * @param combMasks : Output.
 *            Masks where the sum of distances for the sides
 *            marked as 1 are lower than the threshold.
 *
 * @param combDists : Output.
 *            The accumulated sum of distances for each mask
 *
 * @return The length of combMasks
 */
int findNeighborMasks(float* sqrdDists,
                      int nPlanes, float threshold,
                        int* combMasks, float* combDists)
{
    // Add single values below the threshold to lists
    int setLen = 0;
    for (int i = 0; i < nPlanes; i++) {
        if (sqrdDists[i] < threshold) {
            combMasks[setLen] = 1 << i;
            combDists[setLen] = sqrdDists[i];
            setLen++;
        }
    }

    // For every value starting with the initial setLen, try
    // to combine the value with all other values that comes
    // after it in the list can be combined
    // if combDists[k] + combDists[i] < threshold
    int underThreshLen = setLen;
    for (int k = underThreshLen - 2; k >= 0; k--) {
        int tmp = setLen;
        for (int i = k + 1; i < setLen; i++) {
            if (combDists[k] < threshold) {
                int dist = combDists[k] + combDists[i];
                if (dist < threshold) {
                    combMasks[tmp] =
                        combMasks[k] | combMasks[i];
                    combDists[tmp] = dist;
                    tmp++;
                }
            }
        }
        setLen = tmp;
    }

    return setLen;
}
```

Some last minute comments may be made about the efficiency of this implementation.[2]

After execution is finished, each mask in the finished mask-array 'combMasks' represents combinations of hyperplanes whose border is closer to the point than the given threshold. The indices for these boxes themselves can be found by taking the bitwise xor for all the elements in the set with the hashcode.

[2]I noticed this way too late to change it and still be able to re-test the code in time, but the two implementations of the algorithm seem to be implemented inefficiently. In the inner for-loop, the algorithm checks if the combined distance of the k'th iteration is less than the threshold for every iteration; however, the value of neither combDists[k] nor threshold changes inside the inner loop, so this only has to be done once per iteration of the outer loop, and if the distance is greater than the threshold the entire inner loop can be skipped. The algorithm should therefore have the same functionality, but likely better performance, if the if-statement is moved outside the for-loop. The program was tested with the implementation shown above however.

6.6 Creating Good Orthogonal Hyperplanes

Since the method I have introduced in this chapter only requires a single LSH table, investment into finding good orthogonal hyperplanes suddenly becomes much more relevant and worthwhile. With only one LSH table, one does not have to worry about the hyperplanes being sufficiently different between tables (as they would have to be for the multi-table solution), it is enough to find one good set of hyperplanes where the hyperplanes together partition the search space as effectively as possible, one does not need to make sure the LSH tables themselves partition the search space in substantially different ways since there is only a single table; this makes the task of finding "good" hyperplanes more manageable.

As mentioned earlier, for this algorithm to work one needs orthogonal hyperplanes. The way I first implemented the algorithm to construct the hyperplanes was that I made an algorithm that created many random hyperplanes, made them orthogonal and made their elements sum to zero, then I created a score based on how evenly a hyperplane split the search space and how far away from each hyperplane each point was. This alone did not create good hyperplanes, it seemed that if one chose hyperplanes based on their performance to this cost criteria alone, the hyperplanes tended to split the search space in similar ways (grouping the same points together), this happened even though the hyperplanes were made to be orthogonal, which indicates that the SIFT feature descriptors are likely clustered in space along several dimensions (this is also an indication that there may be something to be gained by performing dimensionality reduction on the data, though I will not do so in this thesis).

The algorithm I instead decided to make is a bit more advanced. It creates good hyperplanes by iteratively adding good hyperplanes that correspond well to each other to a list. For each iteration, the algorithm creates many candidate hyperplanes, then it measures how well each new potential hyperplane helps split the search space. Each hyperplane receives a score depending partially on how well it is able to split the (already partially partitioned) search space in cooperation with the other hyperplanes already added to the list of good hyperplanes. The score for each new hyperplane also depends on the distance it has from all the points in the training set.

The algorithm I came up with was the following:

```
nPoints = 100000;
vectorLength = 128;

% the points to fit the hyperplanes to
points = fvecs_read("../test_data/sift/sift_learn.fvecs",
                    nPoints);
points = points';

maxPlanes = 20;
hyperplanes = zeros(maxPlanes, vectorLength);

nAlternatives = 100;
```

```
splitScore = zeros(1, nAlternatives);
absDists = zeros(1,nAlternatives);

groupSizes = zeros(1,1);
groupSizes(1) = nPoints;
groupArray = 1:nPoints;
groupArray2 = zeros(1,nPoints); % Temporary array
whichSide = zeros(1,nPoints); % Temporary array

for nPlanes = 1:maxPlanes
    fprintf("Finding hyperplane %d of %d\n",
            nPlanes, maxPlanes);

    % Get alternatives as well as some extra to adjust them
    alts = 2*rand(2*nAlternatives, vectorLength) - 1;

    % Make them orthogonal to the hyperplanes
    for i = 1:(2*nAlternatives)
        for j = 1:(nPlanes-1)
            alts(i,:) = alts(i,:) - ...
                (alts(i,:) * hyperplanes(j,:)') ...
                    * hyperplanes(j,:);
        end
    end

    % Make them all sum to zero
    % (Only a good idea if the points are all positive as is
    %  the case for the actual sift descriptors.
    %  It would not be necessary if one could try even more
    %  hyperplanes, but might shorten the search)
    for i = 1:nAlternatives
        alts(i,:) = alts(i,:) ...
      - ((sum(alts(i,:)) / sum(alts(nAlternatives+i,:))) ...
       * alts(nAlternatives+i,:));
    end

    % Normalize vectors
    for i = 1:nAlternatives
        alts(i,:) = alts(i,:) ...
            / sqrt(alts(i,:) * alts(i,:)');
    end

    % Create scores for how well it separates the search
    % space and for average distance between point and
    % hyperplane
    nGroups = 2^(nPlanes-1);

    for i = 1:nAlternatives
        splitScore(i) = 0;
        idx = 1;
        for j = 1:nGroups
            positiveSide = 0;
            for k = idx:(idx+groupSizes(j)-1)
```

```
                prod = alts(i,:) * points(groupArray(k),:)';
                if (prod > 0)
                    positiveSide = positiveSide + 1;
                end
                absDists(i) = absDists(i) + abs(prod);
            end
            negativeSide = groupSizes(j) - positiveSide;
            splitScore(i) = splitScore(i) + ...
                            min(negativeSide, positiveSide);

            idx = idx + groupSizes(j);
        end
    end

    splitScore = splitScore / max(splitScore);

    absDists = absDists / max(absDists);

    score = absDists + splitScore;

    % Add hyperplane with best scores to the hyperplane list
    [minimum_score, bestIdx] = max(score);

    hyperplanes(nPlanes,:) = alts(bestIdx,:);

    % Recreate groupSizes and groupArray
    hplane = hyperplanes(nPlanes,:);
    nGroups2 = 2*nGroups;
    groupSizes2 = zeros(1,nGroups2);

    idx = 1;
    for i = 1:nGroups
        for k = idx:(idx+groupSizes(i)-1)
            prod = hplane * points(groupArray(k),:)';
            if (prod <= 0)
                groupSizes2(2*i-1) = groupSizes2(2*i-1) + 1;
                whichSide(k) = 1;
            else
                groupSizes2(2*i) = groupSizes2(2*i) + 1;
                whichSide(k) = 2;
            end
        end
        nextIdx1 = idx;
        nextIdx2 = idx + groupSizes2(2*i-1);
        for k = idx:(idx+groupSizes(i)-1)
            if (whichSide(k) == 1)
                groupArray2(nextIdx1) = groupArray(k);
                nextIdx1 = nextIdx1 + 1;
            else
                groupArray2(nextIdx2) = groupArray(k);
                nextIdx2 = nextIdx2 + 1;
            end
        end
        idx = idx + groupSizes(i);
    end

    groupArray = groupArray2;
    groupSizes = groupSizes2;
end

fprintf("Finished finding hyperplanes\n");
```

Before any hyperplane is considered they will all be normalized and their elements will be made to sum to zero.

The algorithm starts with an empty set of hyperplanes. For the first hyperplane it is checked how well it partitions the search space alone, measured by two criteria: how equal the number of points to either side of the hyperplane is, and the average distance between the points and the hyperplane.

The algorithm starts off with an LSH table without any hyperplanes, corresponding to a single large box with all the points in it, then counting how many points fall on the side of the hyperplane with the least amount of points (this is a score which represents how equally the points are distributed to both sides of the hyperplane), the other score is found by simply combining the absolute distances from the points to the hyperplane. The hyperplane with the best score is chosen.

When adding more hyperplanes, finding the average distance is still done largely the same way. The algorithm does not measure how equally the points in the training set are distributed to each side of only the new hyperplane anymore, instead it measures how well the points that got grouped together in the same group by all the previous hyperplanes are now divided into different boxes by the current hyperplane. This is done by experimenting with splitting each box in the existing LSH table in two, then accumulating the number of points in the smallest subdivision for each of the boxes and accumulating all these numbers into a single value.

The distances from the points to the hyperplane are still used and combined with the other score that indicates how equally the points were distributed. Again, these two scores are combined into a single score which will measure how well the hyperplane splits the search space in combination with the hyperplanes already in the table. The algorithm will then choose the best hyperplane by who has the highest combined score.

After the best hyperplane is found, the algorithm will then actually split every box in the LSH table into two smaller pieces, then combine all these pieces to make an LSH table with a more finely partitioned search space (now also partitioned by the new hyperplane in addition to the older ones). As one can see from the code, there is some intricate logic to make this happen, the algorithm creates two subdivisions of each individual box in the LSH table and moves points into their respective subdivision depending on which side of the hyperplane they fall on, these subdivisions become the new boxes/groups of a more finely partitioned LSH table. Alternatively, one can also recreate a new LSH table from scratch, this would increase the time required by the algorithm, but the time difference might be negligible compared to the time taken to find a good hyperplane in the first place.

What is different between this and simply measuring the hyperplanes one by one is that the algorithm here makes sure that the performance of each hyperplane is measured in relation to all the previous hyperplanes in the LSH table, thus a hyperplane that correspond particularly well with the other hyperplanes already in the list will be added. The idea is that the hyperplanes will work better as a team, even though each individual hyperplane might not optimally split the search space in two halves when used in isolation (except perhaps the first one).

6.7 Function Explanation and Optimization

The single-table version of the algorithm is split into the same general steps as the multi-table version; how each step works however, is a little different.

First, when calculating the hash values for the query points, the distances also has to be calculated; other than that calculating hash values is simplified as it only has to be done for a single set of hyperplanes. This means that there is less to optimize from the memory movement side, since the `calculate_hash_values` function is not called in a loop as it was for the multi-table version of LSH.

Organizing points into groups is done roughly the same way as before, except that this time I decided to discard the size array after creating the index array and instead add the length of `groupArray` (the number of boxes) as the last element in the index array. This was simply a different design choice that re-implementing the algorithm allowed me to make. This provides minor changes to the match_points function.

Finding potential matches is the main way that this algorithm deviates from the previous one. The boxes to check are found the way I described earlier: multiple tables are not used in order to check multiple boxes, instead nearby boxes are checked using the same table. This way of finding nearby boxes was described in Section 6.3.

The point matching phase could be exactly the same as before, but I did make some changes to how it works. I changed the way the algorithm finds the length of the boxes by comparing the current index to the next index in the index array instead of checking the size of the box in a separate array. There are also other changes, the arrays for storing the best and second best matches, as well as the arrays for storing the corresponding distances, have their default values filled in on the device side and are therefore not copied from host to device as they were before. There are also other minor changes to the program. The performance difference for all these changes should, in theory, be relatively low.

The optimizing stages for the `match_points` function will be very similar as before. I will therefore only show the last step of the optimization, when everything is fully optimized.

The function for calculating hash values really is exactly the same as before, except that the last optimization stage is cut out since it's not necessary anymore. I will therefore not explain the hashing part of the algorithm here either.

The function for calculating hash values and distances for the query point was not optimized at all, and the differences in how it is implemented is small. That function will also not be covered.

6.7.1 Constructing LSH Tables

As previously mentioned, this part of the algorithm is almost the same as it was for the multi-table implementation of LSH. The only major difference is that only a single table is used. I also added a minor change where the number of points was added as the last element to `groupIndexMap`, which makes it possible to discard the `groupSizeMap` array if one decides to do so.

6.7.2 Finding Potential Matches

```
/**
 * For every query point, the function goes through all the
 * groups in groupArray that said point can get mapped into.
 * groupArray already stores these points as indices into
 * the base point array, it does this for all tables.
 * Then the function puts the indices belonging to these
 * base points into the potentialMatches array.
 *
 * While creating the potentialMatches array, the function
 * creates and maintains two other arrays called
 * potentialMatchesIndices which is used to index
 * the potentialMatches array.
 *
 * @param nPoints: Number of query points.
 *
 * @param indexGroupMap : Calculated hash values for
 *                        query points.
 *
 * @param groupArray : Arrays with point indices sorted by
 *                     their group.
 *
 * @param groupIndexMap : Arrays with group indices,
 *                        indexes into groupArray.
 *
 * @param pPotentialMatches : Output.
 *                Pointer to an array with indices into the
 *                base point array containing the indices of
 *                all possible matches for each query point
 *
 * @param pPotentialMatchesIndices : Output.
 *                Pointer to array with indices into
 *                potentialMatches
 *
 * @return The total amount of potential matches
 */
int find_potential_matches(
    int nPlanes,
    int nGroups,
    int nPoints,
    int* __restrict__ indexGroupMap,
    float* __restrict__ sqrdDists,
    int* __restrict__ groupArray,
    int* __restrict__ groupIndexMap,
    int** __restrict__ pPotentialMatches,
    int** __restrict__ pPotentialMatchesIndices)
{
    int maxPotentialMatchesLen = 1e9;
    int* potentialMatches =
        (int*) malloc(maxPotentialMatchesLen * sizeof(int));
    int* potentialMatchesIndices =
        (int*)malloc((nPoints + 1) * sizeof(int));

    // Masks and distances for all combinations of
    // hyperplanes where the euclidean distance from the
    // point to the intersection of the hyperplane is below
    // a set threshold.
    // nGroups is the theoretical upper limit of possible
    // combinations, but in reality the number of potential
    // matches will likely never be that high
    int* combMasks = (int*)malloc((1+nGroups)*sizeof(int));
    combMasks[0] = 0;
    float* combDists =(float*)malloc(nGroups*sizeof(float));
    int cnt = 0;
    for (int i = 0; i < nPoints; i++) {
        // Set the index to the groups that the query point
        // is mapped to
        potentialMatchesIndices[i] = cnt;
        // Get the hashcode for this query point
        int hashcode = indexGroupMap[i];
        // Find all combinations of hyperplanes that is
        // closer than sqrt(THRESHOLD)
        int setLen = findNeighborMasks(
            &sqrdDists[i * nPlanes],
            nPlanes, THRESHOLD,
            combMasks+1,
            combDists);
        // Find potential matches by checking hashcode and
        // all hashcodes within a certain distance
        for (int j = 0; j < setLen+1; j++) {
            // get next hashcode
            int hc = hashcode ^ combMasks[j];
            // Find start and end index of the group
            int kStart = groupIndexMap[hc];
            int kEnd = groupIndexMap[hc+1];
            int groupSize = kEnd - kStart;
            // Check if we have reserved enough memory
            if (cnt + groupSize >= maxPotentialMatchesLen) {
                fprintf(stderr, "Not␣enough␣memory.\n");
                exit(1);
            }
            // Add point indices in group hc to
            // potentialMatches
            for (int k = kStart; k < kEnd; k++) {
                potentialMatches[cnt] = groupArray[k];
                cnt++;
            }
        }
    }
    potentialMatchesIndices[nPoints] = cnt;

    free(combMasks);
    free(combDists);
    *pPotentialMatches = potentialMatches;
    *pPotentialMatchesIndices = potentialMatchesIndices;

    return cnt;
}
```

The function uses a huge 4GB (one billion integers) array to potentially store up to a billion potential matches. It should be said that when matching 10 000 query points with a million base points, each query points only needs to check 10% of the base points (which results in 100 000 base points) before the program runs out of memory, even when this much memory is allocated.

The algorithm also reserves two arrays with a length of `nGroups` which is equal to 2^H where H is the number of hyperplanes. These arrays, `combMasks` and `combDists`, are used to store the masks related to sets of hyperplanes and their accumulated squared distance to the query point respectively. These arrays are refilled for every loop-iteration of the program (for every call to `findNeighborMasks` which is done once per query point).

Most of the memory reserved for `combMasks` and `combDists` is almost never actually used; it is allocated for these arrays because it is the absolute upper limit of how many boxes that can theoretically be checked for any query point, and thereby also an upper limit for how much memory these arrays require in a worst case scenario. `combMasks` also has an extra element of value zero, which is added to simplify the program a little bit later on.

The for-loop then checks for potential values for each query point and stores them in the `potentialMatches` array. The hyperplanes that are crossed to find the boxes that need to be checked are found using the `findNeighborMasks` function, this function was described in Section 6.5 on finding accumulated distances.

The boxes that need to be checked are found by applying binary `xor` operations between the hashcode and each mask found by `findNeighborMasks`. Each base point reference found in any of these boxes are then added to the list of potential matches, in the box dedicated to potential matches for this particular query point.

6.7.3 Matching Points

For the `match_points` function I also only made slight changes, I will present that version of the function here.

```
/**
 * Uses result from find_potential_matches to match
 * query points with base points
 *
 * @param nQueryVecs : Number of query vectors
 *
 * @param nBaseVecs : Number of base vectors
 *
 * @param queryVecs : Query vectors
 *
 * @param baseVecs : Base vectors (Array of points used to
 *                   create the LSH tables)
 *
 * @param nPotentialMatches : Number of potential matches
 *
 * @param potentialMatches : An array with indices into
 *                           point1 containing the indices
 *                           of all possible matches for
 *                           each query point
 *                           (i.e. potential matches between
 *                            query points and base points)
 *
 * @param potentialMatchesIndices : Indices used to index
 *                                  into potentialMatches
 *
 *
 * @param lshMatches : Output.
 *                     Index into base points representing
 *                     the best match between query points
 *                     and base points
 *
 * @param bestMatchDists : Output.
 *                         Distances between points for
 *                         every match in lshMatches
 *
 * @param lshMatches2 : Output.
 *                      Index into base points representing
 *                      the 2nd best match between the
 *                      query points and the base points
 *
 * @param bestMatchDists2 : Output.
 *                          Distances between points for
 *                          every match in lshMatches2
 */
void match_points(
    int nQueryVecs,
    int nBaseVecs,
    float* __restrict__ queryVecs,
    float* __restrict__ baseVecs,
    int nPotentialMatches,
    int* __restrict__ potentialMatches,
    int* __restrict__ potentialMatchesIndices,
    // outputs
    int* __restrict__ lshMatches,
    float* __restrict__ bestMatchDists,
    int* __restrict__ lshMatches2,
    float* __restrict__ bestMatchDists2)
{
#pragma acc data \
    pcopyin(queryVecs[nQueryVecs*vectorLength]) \
    pcopyin(baseVecs[nBaseVecs*vectorLength]) \
    pcopyin(potentialMatches[nPotentialMatches])\
    pcopyin(potentialMatchesIndices[nQueryVecs+1]) \
    pcopyout(lshMatches[nQueryVecs]) \
    pcopyout(bestMatchDists[nQueryVecs]) \
    pcopyout(lshMatches2[nQueryVecs]) \
    pcopyout(bestMatchDists2[nQueryVecs])
 {
    #pragma acc parallel loop gang worker \
        num_workers(1) vector_length(32)
    for (int i = 0; i < nQueryVecs; i++) {
        // Find the group of elements from groupArray to
        // match with
        float bestMatchDist = 1e10;
        float bestMatchDist2 = 1e10;
        int match = -1;
        int match2 = -1;
        int jStart = potentialMatchesIndices[i];
        int jEnd = potentialMatchesIndices[i + 1];

        // Match the points
        #pragma acc loop seq
        for (int j = jStart; j < jEnd; j++) {
            int idx = potentialMatches[j];
            //dif=sum((queryVecs[i][:]-baseVecs[idx][:]).^2)
            float diff = 0;
            #pragma acc loop reduction(+:diff) vector
            for (int k = 0; k < vectorLength; k++) {
                float tmp = queryVecs[i * vectorLength + k]
                        - baseVecs[idx * vectorLength + k];
                diff += tmp * tmp;
            }
            // check if distance is the lowest distance
            // so far
            if (diff < bestMatchDist) {
                // save old value as second best
                bestMatchDist2 = bestMatchDist;
                match2 = match;
                // update best value
                bestMatchDist = diff;
                match = idx;
            }
        }

        lshMatches[i] = match;
        bestMatchDists[i] = bestMatchDist;
        lshMatches2[i] = match2;
        bestMatchDists2[i] = bestMatchDist2;
    }
 }
}
```

I removed an `if` statement before changing the "best match" and "best dist", this was not necessary in the last implementation either, but rather a remnant of how the function used to work, there are however both pros and cons in terms of performance, though these changes in performance should not be very significant in reality.

I also did not set the default best match distance values on the host side to then copy them over to the device side, instead I set the default values in the functions itself. Setting these values within the algorithm allows for simply copying the the "best match" and "best dist" arrays from device to host instead of having to copy these arrays both ways.

As mentioned previously this version of the function does not need the `potentialMatchesLengths` array since it can calculate these values from the indices stored in `potentialMatchesIndices`.

In this version I also specified that the second loop should be performed sequentially, this is assumed by the compiler anyways, but one can specify it if one wants to and for this function I decided to demonstrate this.

Chapter 7

Single Table LSH: Results

7.1 Statistics About the Feature Descriptors

Before going into the results of the single table implementation I will first present some statistics about the SIFT feature descriptors from the dataset that I am using. I am covering these details here since it becomes relevant when setting the threshold for checking the other side of a hyperplane, and thereby becomes relevant in this chapter.

We have the following approximate statistics:

1. The average length of a feature vector is **508.6**. The length of all the feature vectors are approximately equal to each other with very little variation.

2. The average distance from a feature vector to a random hyperplane (of length one and whose elements sum to zero) is around **28** on average, though this will of course vary substantially for different feature vectors and hyperplanes. This is about 5.5% of the euclidean length of the feature vectors.

3. The average distance from a feature vector to one of the improved hyperplanes is very high for the first hyperplane in the list, then rapidly declines and does not vary too much for the other hyperplanes, likely due to other priorities taking precedence over distance between vectors and points when the improved hyperplanes were created. The average distance from a feature vector to one of the improved hyperplanes seems to be around **25** when many hyperplanes are used. The average distance starts from 68 for the first hyperplane, 28 for the second, then varies between 19 and 28 for the rest of the elements (this is just one arbitrary set of the improved hyperplanes that I created).

4. The average distance between two random feature vectors seems to be about **536** on average, which is slightly higher than the length of the feature vectors.

5. The average distance from a feature vector to its best match seems to be about **188**, which is about 37% of the average length of the feature

vectors. This distance is also approximately 35% of the average distance between two arbitrary feature vectors, and it is 6.7 times the average distance from a feature vector to a hyperplane when using random hyperplanes whose elements sum to zero.

For most of the results in this chapter we choose the threshold for the maximum distance from point to hyperplane before checking the other side of the hyperplane to be 35, which is equal to a distance of 6.88% of the euclidean length of the feature vectors.

This threshold will be larger than the majority of the distances from a point to a hyperplane. However, it will also be lower than the majority of the distances from a point to a greater combination of hyperplanes. As we will see, the majority of the search space will not be searched for a single query.

The threshold is also only 18.6% of the average length from a query point to its best match in the base set. This means that the algorithm will do little to guarantee correct matching for the vast majority of the query points. While the threshold does not guarantee matches for most reasonable distances, it still does a good job of searching boxes in the general area of the query point which has a high probability of finding the correct match. So even though the algorithm is theoretically capable of solving the exact nearest neighbor problem, it is here used to find approximate nearest neighbors instead as that is a more practical and realistic use case.

7.2 Performance of Each Optimization

When measuring the performance of the algorithm, the improved hyperplanes (created using the method outlined in Section 6.6) were used. For all these results the threshold for checking the other side of a group of hyperplanes is set to 35 and the number of hyperplanes is 16. This chapter is all tested on the large dataset, the smaller dataset is only used for the last section.

Potential matches found	272585646
Comparisons per query vector	27258.5646
Average portion of search space searched	2.7259%
Correct classification ratio	83.36%

Table 7.1: Details (same for all optimization levels presented)

	Time	**% of total time**
Total time	2.44s	100%
Calculating hash values for base vectors	0.283s	11.6%
Constructing LSH table	0.008s	0.328%
Calculating hash values for query vectors	0.005s	0.205%
Finding potential matches	0.658s	27.0%
Matching potential matches	1.01s	41.4%

Table 7.2: Real run-time for each phase after full optimization. Same table as Table 7.7, but also shown here underneath the information in Table 7.1 to increase readability of the final results.

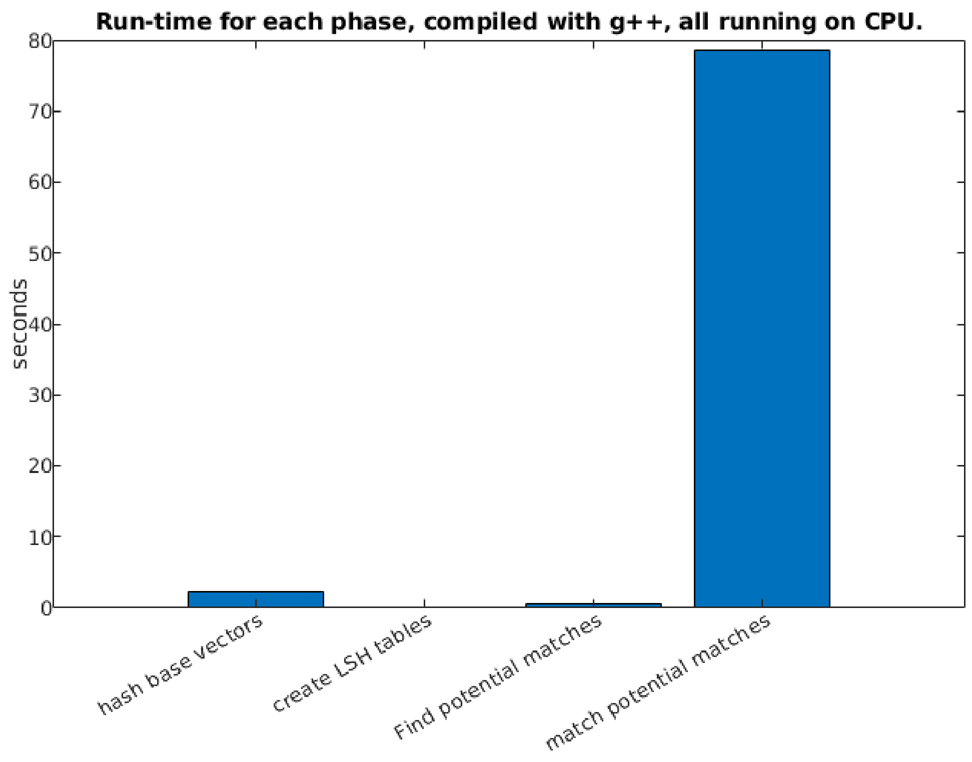
In comparison, for the multi-table LSH version, the correct classification ratio was 83.03% and the average portion of search space searched was only 2.3978%. The correct classification ratio and the average portion of the search space cannot be compared directly, since these statistics will also depend on the number of hyperplanes vs the number of tables for the multi-table version of LSH, or the number of hyperplanes vs the threshold for checking the other side of hyperplanes for the single-table version of LSH. What matters when comparing these two versions of LSH is really the time taken vs. the number of correct matches.

7.2.1 CPU program, g++ compiler

From Table 7.3 we can see that the vast majority of the time is now spent on matching potential matches. For the serial version, calculating hash values and constructing LSH tables are now only minor parts of the program due to the fact that only a single LSH table is used.

	Time	**% of total time**
Total time	81.9s	100%
Calculating hash values for base vectors	2.25s	2.75%
Constructing LSH table	0.009s	0.0110%
Calculating hash values for query vectors	0.025s	0.0305%
Finding potential matches	0.625s	0.763%
Matching potential matches	78.6s	96.0%

Table 7.3: Run-time for each phase, compiled with g++, all running on CPU.

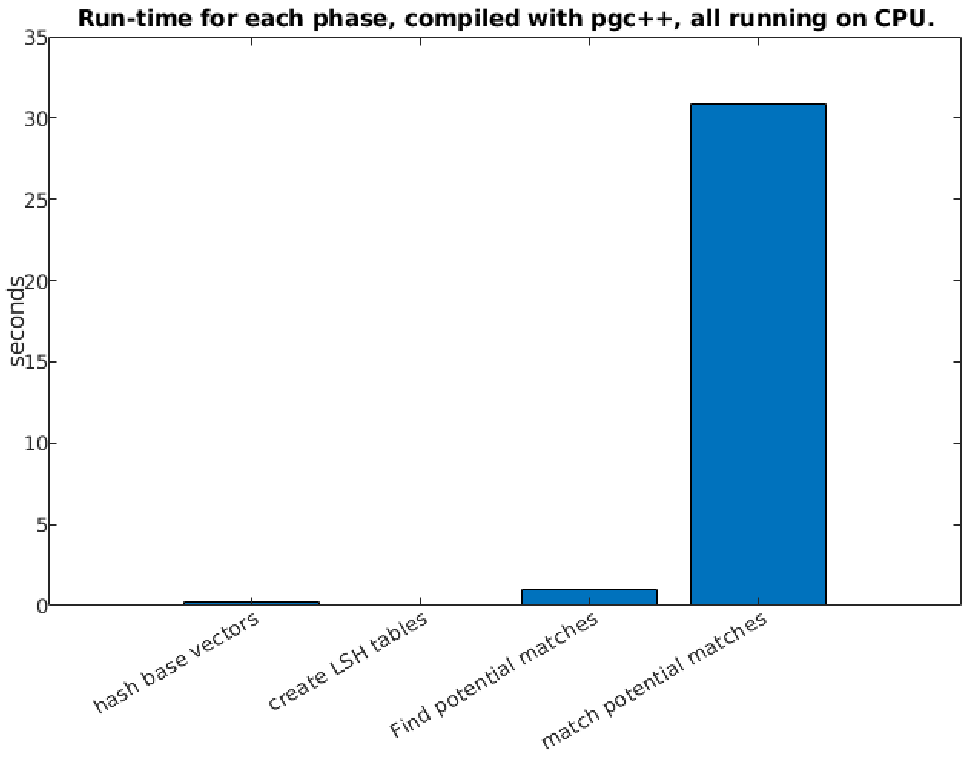


7.2.2 CPU program pgc++ compiler

When using the pgc++ compiler the time spent on both calculating hash values and matching potential matches are rapidly reduced. The time spent matching potential matches had a 2.54x speedup. The time spent hashing the base vectors had a 8.75x speedup.

	Time	**% of total time**
Total time	33.5s	100%
Calculating hash values for base vectors	0.257s	0.767%
Constructing LSH table	0.009s	0.0269%
Calculating hash values for query vectors	0.002s	0.00597%
Finding potential matches	0.966s	2.88%
Matching potential matches	30.9s	92.2%

Table 7.4: Run-time for each phase, compiled with pgc++, all running on CPU.



7.2.3 Parallelized hashing

Parallelizing the hash function seems to slightly increase the time spent due to the gains from GPU acceleration not making up for the overhead associated with it.

	Time	% of total time
Total time	35.6s	100%
Calculating hash values for base vectors	0.305s	0.857%
Constructing LSH table	0.009s	0.0253%
Calculating hash values for query vectors	0.003s	0.00843%
Finding potential matches	1.015s	2.85%
Matching potential matches	33.7s	94.7%

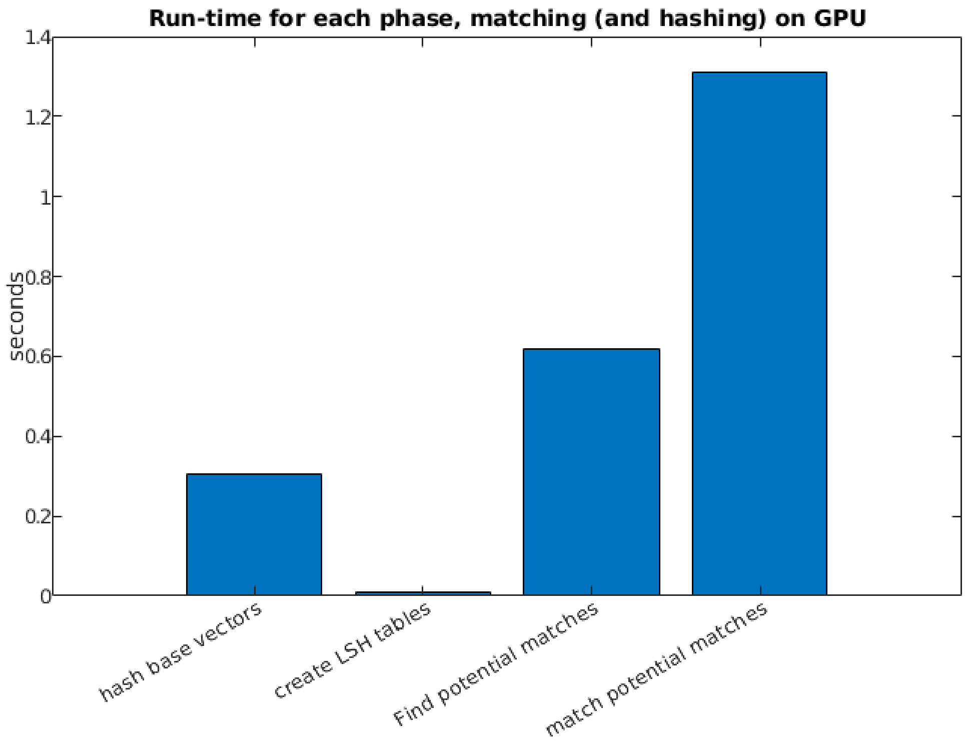
Table 7.5: Run-time for each phase, hashing on GPU.

7.2.4 Parallelized matching

When parallelizing `match_points` we see that the time spent on matching potential matches decreases significantly from 33.7 to 1.31 seconds, a 25.7x speedup.

	Time	**% of total time**
Total time	2.67s	100%
Calculating hash values for base vectors	0.304s	11.4%
Constructing LSH table	0.009s	0.337%%
Calculating hash values for query vectors	0.003s	0.112%
Finding potential matches	0.620s	23.2%
Matching potential matches	1.31s	49.1%

Table 7.6: Run-time for each phase, matching on GPU.



7.2.5 Loop optimization

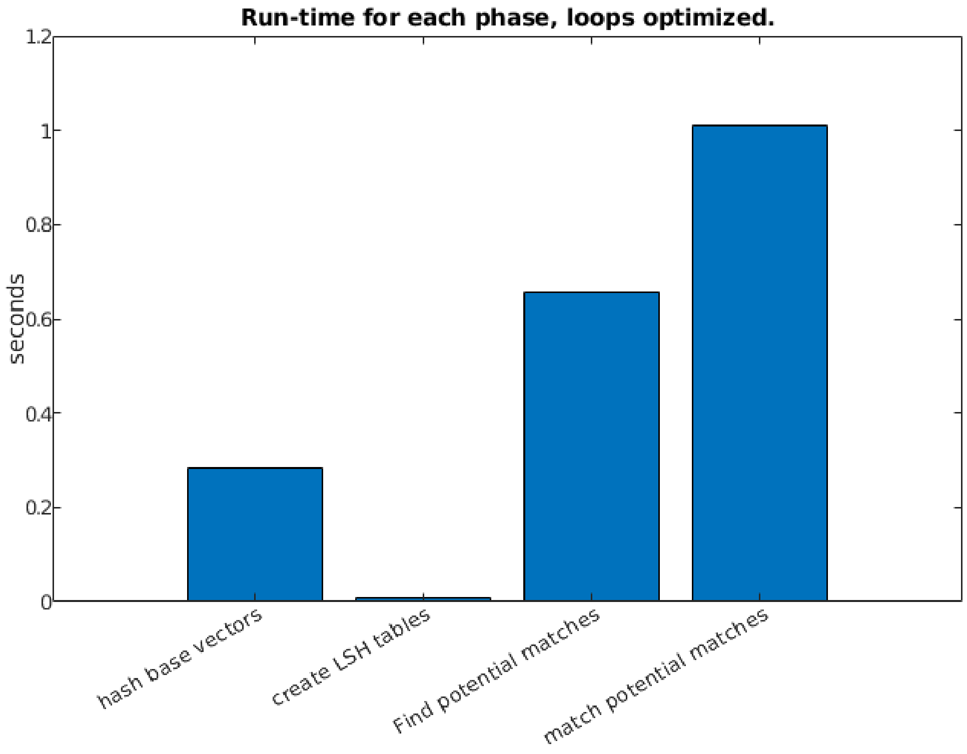
The final optimization is loop optimization, the times for this is shown in the table below (Table 7.7).

Matching points is now running at 1.01 seconds. This results in a 1.30x (30%) speedup from the previous optimization, a 30.6x speedup from the CPU version compiled with the pgc++ compiler (using SIMD instructions), and a 77.8x speedup from the fully serial CPU version compiled with g++.

Calculating hash values for the base vectors now takes 0.283 seconds which is a little more than it did when using the CPU version compiled with the pgc++ compiler which uses SIMD instructions. The end version is still 7.95 times faster than the serial CPU version compiled using the g++ compiler that did not optimize the program with SIMD instructions. One could say that, due to the scale of the problem being smaller, a SIMD version of the program can perform better for this part of the algorithm than the GPU version. The hashing part of the algorithm seems to be of such a medium scale that the time is similar between the CPU and GPU versions (this is due to the algorithm using only one LSH table).

	Time	**% of total time**
Total time	2.44s	100%
Calculating hash values for base vectors	0.283s	11.6%
Constructing LSH table	0.008s	0.328%
Calculating hash values for query vectors	0.005s	0.205%
Finding potential matches	0.658s	27.0%
Matching potential matches	1.01s	41.4%

Table 7.7: Real run-time for each phase hashing and matching on GPU. Program is fully optimized using OpenACC.

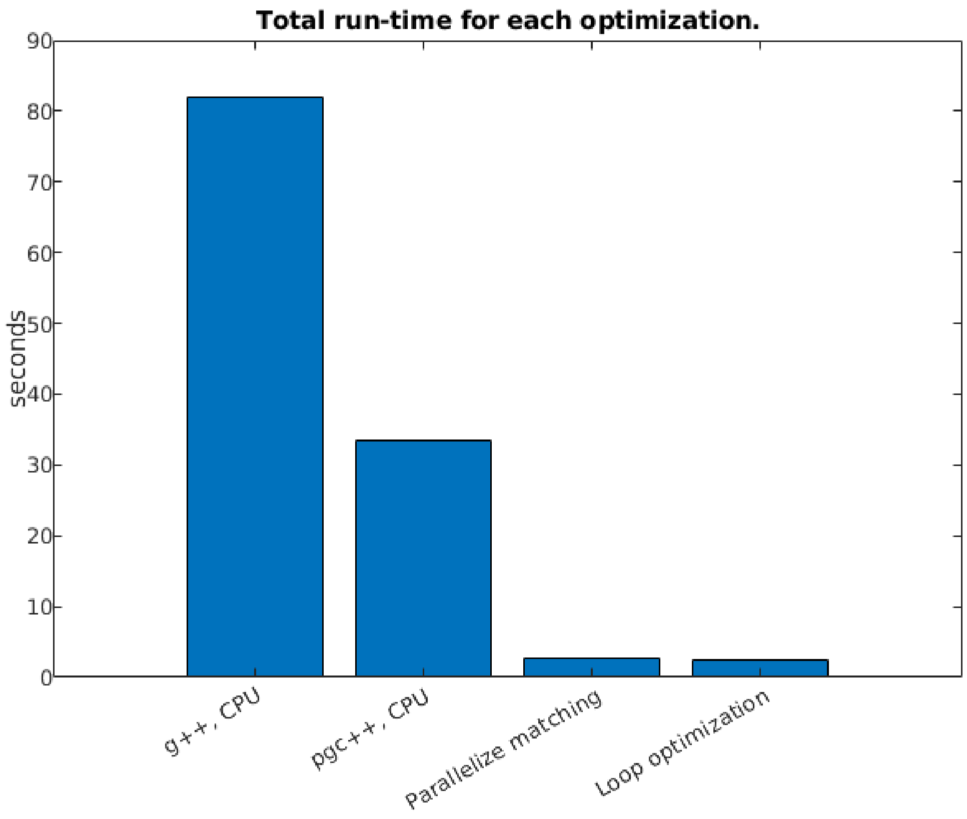


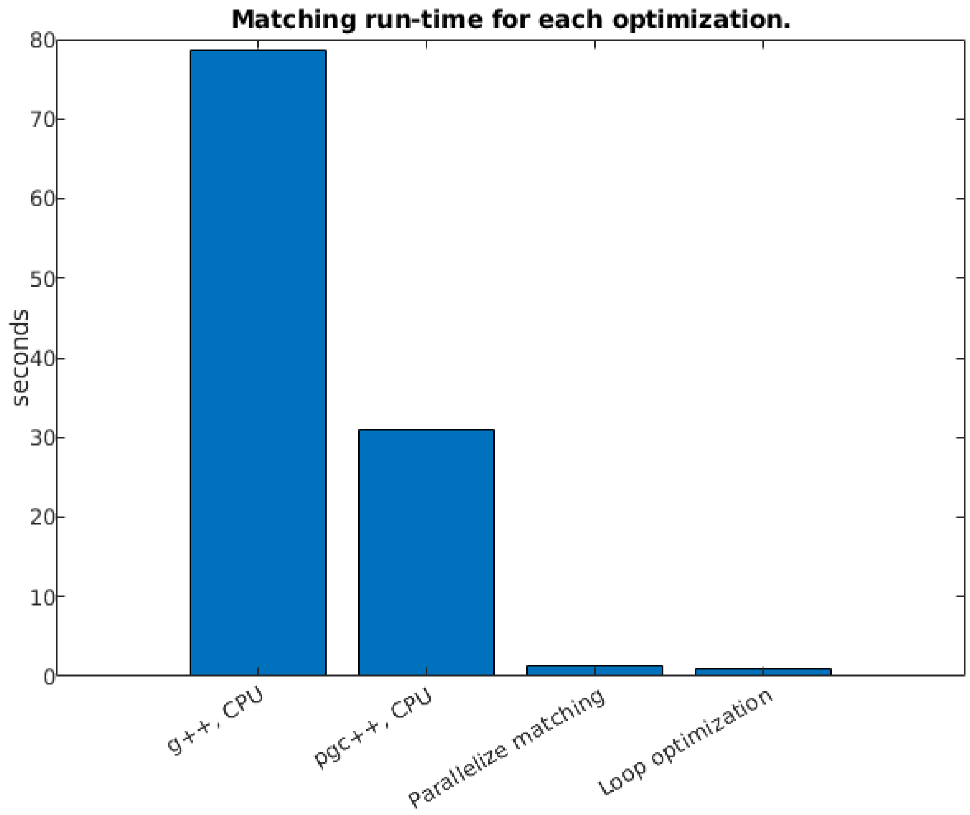
7.3 Performance of Each Optimization Summarized

In this section you can see the improvements in execution time for each optimization stage. In Table 7.8 we see the total time spent by the entire program, as well as the time spent for hashing and the time spent on matching. As before, roughly 0.5 seconds of the total time is spent doing work unrelated to the algorithm but is included since the total time was measured using the `time` command.

	Total time	Hashing base points	Matching
CPU program, g++	81.9s	2.25s	78.6s
CPU program, pgc++	33.5s	0.257s	30.9s
Parallelize	2.67s	0.304s	1.31s
Loop optimization	2.44s	0.283s	1.01s

Table 7.8: Run-time results for each optimization stage.





In Table 7.9 we can see the improvements by each optimization stage of the `match_points` function. Each optimization stage shows what the time of the function is, how much it improved from the CPU version of the program compiled with `pgc++` (both in percentage and speedup), as well as the incremental speedup (how much faster the program runs in this optimization stage than in the last optimization stage).

	Time	**Speedup from CPU (g++)**	**Speedup from CPU (pgc++)**	**Incremental speedup**
CPU, g++	78.6s	1x	-	-
CPU program, pgc++	30.9s	2.54x	1x	2.54x
Parallelize	1.31s	60.0x	23.6x	23.6x
Loop optimization	1.01s	77.8x	30.6x	1.30x

Table 7.9: Matching run-times and speed-ups

7.4 Improved Hyperplanes vs Random Hyperplanes

In this section I will attempt to compare the improved hyperplanes created using the algorithm presented in Section 6.6, with random orthogonal hyperplanes whose elements sum to zero.

For the comparison I used three sets of improved orthogonal hyperplanes whose elements sum to zero, and three sets of random orthogonal hyperplanes whose elements also sum to zero.

Table 7.10 and Table 7.11 are both using the random hyperplanes. Table 7.10 uses a threshold of 35 since it is the same as the threshold used for the improved hyperplanes in Table 7.12. Table 7.11 uses a threshold of 30 since that makes the results more comparable to the results from using the improved sets of hyperplanes from Table 7.12.

By contrasting the results in Table 7.12 with the results in Table 7.10 we can see that the improved set of hyperplanes does in fact partition the search space more equally and effectively. The different sets used the same number of hyperplanes and the threshold was the same, and yet the improved hyperplanes in Table 7.12 partitioned the search space in a way where a much smaller portion of the search space had to be searched.

Whether this indicates that the improved hyperplanes are actually better is hard to compare using these two tables however, since while the random hyperplanes result in having to search more of the search space, they also result in more correct classifications. To make the results more comparable I also added a table of the results for random hyperplanes with a slightly lower threshold in Table 7.11.

By comparing the results from the improved hyperplanes in Table 7.12 with the random hyperplanes with a threshold of 30 in Table 7.11, we see the improved hyperplanes is able to search less of the search space and still get more correct results. This indicates that the improved hyperplanes are generally better than the random hyperplanes, though the difference is perhaps not quite as drastic as one might have hoped.

While the run-times from "finding potential matches" are very variable, they are also quite similar between the two tables which indicates that this increased efficiency does not come at any other cost. Since the improved hyperplanes used in Table 7.12 partition the search space in smaller, more equal pieces, and has to use a higher threshold in order to get a comparable correct ratio, it is actually quite reasonable that it might spend a similar amount of time searching for the boxes in the LSH table for potential matches; each box that needs to be searched will have fewer point references, meanwhile, more boxes will have to be searched since the threshold for not checking a neighboring box is higher.

It seems that if the version using improved hyperplanes and the version using random hyperplanes find the same amount of potential matches, the one with the improved hyperplane will have to spend more time searching boxes for potential matches. However, the version with the improved hyperplanes can get away with searching fewer points overall, which means that the time spent searching boxes for potential matches seems to be similar for both versions to achieve similar correctness results.

	1	2	3
Correct ratio	88.2%	87.1%	87.8%
Average % of search space searched	7.44%	6.02%	6.65%
Total time	3.669s	2.894s	4.404s
Calculating hash values for base vectors	0.274s	0.266s	0.275s
Constructing LSH table	0.007s	0.006s	0.008s
Calculating hash values for query vectors	0.003s	0.003s	0.004s
Finding potential matches	1.098s	0.798s	1.581s
Matching potential matches	1.731s	1.458s	1.918s

Table 7.10: Statistics and run-times for different sets of random orthogonal hyperplanes that sum to zero. The columns represent different sets of hyperplanes and the rows represent time taken for different parts of the program. Number of hyperplanes was 16, threshold was 35.

	1	2	3
Correct ratio	82.0%	80.4%	81.8%
Average % of search space searched	5.24%	4.21%	4.64%
Total time	3.534s	2.822s	2.407s
Calculating hash values for base vectors	0.290s	0.264s	0.278s
Constructing LSH table	0.010s	0.007s	0.008s
Calculating hash values for query vectors	0.005s	0.002s	0.004s
Finding potential matches	1.230s	0.960s	0.603s
Matching potential matches	1.423s	1.046s	1.004s

Table 7.11: Statistics and run-times for different sets of random orthogonal hyperplanes that sum to zero. The columns represent different sets of hyperplanes and the rows represent time taken for different parts of the program. Number of hyperplanes was 16, threshold was 30.

	1	2	3
Correct ratio	83.4%	82.2%	83.4%
Average % of search space searched	2.73%	2.59%	2.73%
Total time	2.919s	2.592s	2.911s
Calculating hash values for base vectors	0.274s	0.266s	0.267s
Constructing LSH table	0.010s	0.009s	0.009s
Calculating hash values for query vectors	0.003s	0.003s	0.003s
Finding potential matches	0.967s	0.881s	0.945s
Matching potential matches	1.128s	0.910s	1.157s

Table 7.12: Statistics and run-times for different sets of improved orthogonal hyperplanes that sum to zero. The columns represent different sets of hyperplanes and the rows represent time taken for different parts of the program. Number of hyperplanes was 16, threshold was 35.

7.5 Different Numbers of Hyperplanes, Improved Hyperplanes

In this section I present some statistics and run-times for the single table LSH program with different numbers of improved hyperplanes, holding the threshold constant.

Number of hyperplanes	Total time taken	Correct ratio	Average % of search space searched
10	4.169s	86.0%	6.97%
12	3.089s	82.5%	4.33%
14	2.190s	78.7%	2.59%
16	1.961s	75.0%	1.66%
18	1.807s	71.8%	1.15%
20	1.642s	67.8%	0.761%

Table 7.13: Results for different numbers of hyperplanes using the improved set of hyperplanes, and threshold = 30.

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH table	Finding potential matches	Matching potential matches
10	0.271s	0.005s	1.140s	2.195s
12	0.272s	0.006s	0.953s	1.302s
14	0.300s	0.006s	0.468s	0.897s
16	0.265s	0.009s	0.546s	0.642s
18	0.262s	0.013s	0.577s	0.458s
20	0.265s	0.018s	0.619s	0.385s

Table 7.14: Run-times for each part of the program with different numbers of hyperplanes using the improved set of hyperplanes, and threshold = 30.

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH table	Finding potential matches	Matching potential matches
10	6.50%	0.120%	27.3%	52.7%
12	8.81%	0.194%	30.9%	42.1%
14	13.7%	0.274%	21.4%	41.0%
16	13.5%	0.459%	27.8%	32.7%
18	14.5%	0.719%	31.9%	25.3%
20	16.1%	1.10%	37.7%	23.4%

Table 7.15: Percentage of total time for each part of the program with different numbers of hyperplanes using the improved set of hyperplanes, and threshold = 30.

7.6 Different Numbers of Hyperplanes, Random Hyperplanes

In this section I present some statistics and run-times for the single table LSH program with different numbers of random hyperplanes, holding the threshold constant.

In order to again compare the random hyperplanes to the improved ones, one can compare the row with 18 random hyperplanes in Figure 7.16 with the row with 14 improved hyperplanes in Figure 7.13.

The two versions in that case have similar correct ratios, but different times: the one with the 14 improved hyperplanes took 2.190 seconds, searched 2.59% of the search space, with a correct ratio of 78.7%; the one with the 18 random hyperplanes took 2.644 seconds, searched 3.51% of the search space, and had a correct ratio of 78.6%. Again the program with improved hyperplanes outperformed the program with random hyperplanes, as it had a slightly better correct ratio and finished executing in a smaller amount of time.

Number of hyperplanes	Total time taken	Correct ratio	Average % of search space searched
14	4.002s	84.0%	6.62%
16	3.353s	82.0%	5.24%
18	2.644s	78.6%	3.51%
20	2.462s	76.0%	2.60%

Table 7.16: Results for different numbers of hyperplanes using the random set of hyperplanes, and threshold = 30.

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH table	Finding potential matches	Matching potential matches
14	0.269s	0.006s	1.391s	1.757s
16	0.271s	0.007s	1.137s	1.370s
18	0.267s	0.010s	0.921s	0.908s
20	0.268s	0.019s	0.884s	0.772s

Table 7.17: Run-times for each part of the program with different numbers of hyperplanes using the random set of hyperplanes, and threshold = 30.

Number of hyperplanes	Calculating hash values for base vectors	Constructing LSH table	Finding potential matches	Matching potential matches
14	6.72%	0.150%	34.8%	43.9%
16	8.08%	0.209%	33.9%	40.9%
18	10.1%	0.378%	34.8%	34.3%
20	10.9%	0.717%	35.9%	31.4%

Table 7.18: Percentage of total time for each part of the program with different numbers of hyperplanes using the random set of hyperplanes, and threshold = 30.

7.7 Different Thresholds, Improved Hyperplanes

In this section I present some statistics and run-times for the single table LSH program with different thresholds using improved hyperplanes, holding the number of hyperplanes constant.

Distance threshold	**Total time taken**	**Correct ratio**	**Average % of search space searched**
0	0.913s	6.59%	0.0095%
5	0.862s	15.9%	0.0371%
10	0.956s	27.7%	0.104%
15	1.025s	40.3%	0.242%
20	1.197s	53.1%	0.502%
25	1.465s	65.1%	0.950%
30	1.992s	75.0%	1.66%
35	2.866s	83.4%	2.73%

Table 7.19: Results for different thresholds using the improved set of hyperplanes, and number of hyperplanes = 16.

Distance threshold	**Calculating hash values for base vectors**	**Constructing LSH table**	**Finding potential matches**	**Matching potential matches**
0	0.268s	0.010s	0.003s	0.091s
5	0.258s	0.009s	0.010s	0.101s
10	0.280s	0.008s	0.019s	0.147s
15	0.272s	0.009s	0.072s	0.165s
20	0.263s	0.010s	0.157s	0.261s
25	0.270s	0.009s	0.309s	0.367s
30	0.267s	0.009s	0.574s	0.625s
35	0.273s	0.009s	0.941s	1.117s

Table 7.20: Run-times for each part of the program with different thresholds using the improved set of hyperplanes, and number of hyperplanes = 16.

Distance threshold	**Calculating hash values for base vectors**	**Constructing LSH table**	**Finding potential matches**	**Matching potential matches**
0	29.4%	1.10%	0.329%	9.97%
5	29.9%	1.04%	1.16%	11.7%
10	29.2%	0.941%	1.99%	15.4%
15	26.5%	0.878%	7.02%	16.1%
20	22.0%	0.912%	14.3%	23.8%
25	18.4%	0.614%	21.1%	25.1%
30	13.4%	0.452%	28.8%	31.4%
35	9.52%	0.314%	32.8%	39.0%

Table 7.21: Percentage of total time for each part of the program with different thresholds using the improved set of hyperplanes, and number of hyperplanes = 16.

7.8 Different Thresholds, Random Hyperplanes

In this section I present some statistics and run-times for the single table LSH program with different thresholds using random hyperplanes, holding the number of hyperplanes constant.

Distance threshold	**Total time taken**	**Correct ratio**	**Average % of search space searched**
0	0.917s	10.4%	0.0940%
5	1.007s	22.2%	0.278%
10	1.171s	36.1%	0.630%
15	1.494s	49.9%	1.24%
20	1.890s	62.7%	2.16%
25	2.479s	72.6%	3.49%
30	3.336s	82.0%	5.24%
35	4.450s	88.2%	7.44%

Table 7.22: Results for different thresholds using the random set of hyperplanes, and threshold = 35.

Distance threshold	**Calculating hash values for base vectors**	**Constructing LSH table**	**Finding potential matches**	**Matching potential matches**
0	0.277s	0.006s	0.011s	0.126s
5	0.264s	0.007s	0.059s	0.182s
10	0.262s	0.007s	0.134s	0.266s
15	0.270s	0.006s	0.264s	0.447s
20	0.272s	0.006s	0.468s	0.619s
25	0.267s	0.008s	0.774s	0.894s
30	0.263s	0.008s	1.176s	1.330s
35	0.261s	0.007s	1.660s	1.943s

Table 7.23: Run-times for each part of the program with different thresholds using the random set of hyperplanes, and threshold = 35.

Distance threshold	**Calculating hash values for base vectors**	**Constructing LSH table**	**Finding potential matches**	**Matching potential matches**
0	30.2%	0.654%	1.20%	13.7%
5	26.2%	0.695%	5.86%	18.1%
10	22.4%	0.598%	11.4%	22.7%
15	18.1%	0.402%	17.7%	29.9%
20	14.4%	0.318%	24.8%	32.8%
25	10.8%	0.323%	31.2%	36.1%
30	7.88%	0.240%	35.3%	39.9%
35	5.87%	0.157%	37.3%	43.7%

Table 7.24: Percentage of total time for each part of the program with different thresholds using the random set of hyperplanes, and threshold = 35.

7.9 Small Dataset Results

Like what I did for the multi-table LSH solution, I will also test this solution for smaller dataset. I will use the the same threshold (35) both when using 16 hyperplanes and when using 8 hyperplanes.

When using 16 hyperplanes:

Potential matches found	26369
Comparisons per query vector	263.69
Average portion of search space searched	2.64%
Correct classification ratio	74.0%

Table 7.25: Statistics about the program. Run on the small dataset. 16 hyperplanes used. Threshold = 35. The details are the same for all optimization levels presented.

For the large dataset we were able to get an 83.4% correct ratio while only searching approximately 2.73% of the search space. For the smaller dataset we get a 74.0% correct classification ratio while searching 2.64% of the search space. For the smaller dataset, the correct ratio is quite a bit lower, whereas the average portion of the search space searched is only a little smaller (in theory these results should be the same since the chance for a query point to be separated from its best match should be equal for the two datasets, but differences in the datasets probably result in this difference).

	Time	% of total time
Total time	0.045s	100%
Calculating hash values for base vectors	0.028s	62.2%
Constructing LSH tables	0.001s	3.57%
Calculating hash values for query vectors	0.000s	∼0%
Finding potential matches	0.001s	3.57%
Matching potential matches	0.005s	∼0%

Table 7.26: Run-time for each phase, compiled with g++, all running on CPU. Number of hyperplanes = 16, threshold = 35.

	Time	% of total time
Total time	0.018s	100%
Calculating hash values for base vectors	0.002s	11.1%
Constructing LSH tables	0.001s	5.55%
Calculating hash values for query vectors	0.000s	∼0%
Finding potential matches	0.001s	5.55%
Matching potential matches	0.001s	5.55%

Table 7.27: Run-time for each phase, compiled with pgc++ (using SIMD parallelization), all running on CPU. Number of hyperplanes = 16, threshold = 35.

	Time	% of total time
Total time	0.230s	100%
Calculating hash values for base vectors	0.174s	75.6%
Constructing LSH tables	0.001s	0.435%
Calculating hash values for query vectors	0.000s	∼0%
Finding potential matches	0.001s	0.435%
Matching potential matches	0.002s	0.870%

Table 7.28: Run-time when "Calculating hash values for base vectors" and "Matching potential matches" are run on GPUs and the program is fully optimized. Number of hyperplanes = 16, threshold = 35.

In the fully serial version, see that the time spent by the program is dominated by the hashing of the base points; however, this imbalance is much smaller than what was the case when multiple tables were used, since the hashing is only done for a single table.

When SIMD instructions are used we can see that the time spent hashing points is almost the same in scale as that spent on other individual phases of the program.

Applying the GPU significantly increases the time spent for hashing points, which is due to overhead.

When using 8 hyperplanes:

Potential matches found	142943
Comparisons per query vector	1429.43
Average portion of search space searched	14.3%
Correct classification ratio	91.0%

Table 7.29: Statistics about the program. Run on the small dataset. 8 hyperplanes used. Threshold = 35. The details are the same for all optimization levels presented

	Time	**% of total time**
Total time	0.045s	100%
Calculating hash values for base vectors	0.011s	24.4%
Constructing LSH tables	0.000s	∼0%
Calculating hash values for query vectors	0.000s	∼0%
Finding potential matches	0.001s	2.22%
Matching potential matches	0.024s	53.3%

Table 7.30: Run-time for each phase, compiled with g++, all running on CPU. Number of hyperplanes = 8, threshold = 35.

	Time	**% of total time**
Total time	0.019s	100%
Calculating hash values for base vectors	0.001s	5.26%
Constructing LSH tables	0.000s	∼0%
Calculating hash values for query vectors	0.000s	∼0%
Finding potential matches	0.001s	5.26%
Matching potential matches	0.005s	26.3%

Table 7.31: Run-time for each phase, compiled with pgc++ (using SIMD parallelization), all running on CPU. Number of hyperplanes = 8, threshold = 35.

	Time	**% of total time**
Total time	0.258s	100%
Calculating hash values for base vectors	0.183s	70.9%
Constructing LSH tables	0.001s	0.388%
Calculating hash values for query vectors	0.000s	$\sim$0%
Finding potential matches	0.000s	$\sim$0%
Matching potential matches	0.005s	1.94%

Table 7.32: Run-time when "Calculating hash values for base vectors" and "Matching potential matches" are run on GPUs and the program is fully optimized. Number of hyperplanes = 8, threshold = 35.

When using only 8 hyperplanes, the correct classification ratio increases from 74.0% to 91.0%, meanwhile the average portion of the search space that is searched increases from 2.67% to 14.3%.

For the serial versions of the code, the time spent hashing roughly halves (which is reasonable since the number of hyperplanes is also halved). Meanwhile, the time spent matching potential matches increased by roughly a factor of five, which also makes sense since the average amount of the search space that had to be searched also increased by roughly a factor of five.

For the GPU version, the time spent calculating hash values surprisingly increases slightly (I'm assuming this is mostly random, since the timing is not completely stable). Meanwhile the time spent matching points increases which is to be expected, it increases by roughly a factor of two. For such a small scale problem like this the run-times are largely affected by the overhead associated with GPUs, so the GPU version is much slower than the other two versions both whether we are using many or few hyperplanes.

For the single table version of LSH, and for this small computation scale, the CPU versions outperform the GPU version, whether SIMD operations are used or not. The problem has to be of a certain scale for GPU computing to be effective, and when only a single table is used, the scale of the problem effectively reduces even further for a small dataset, making a GPU implementation less effective compared to CPU implementations.

Part III

Conclusion

Chapter 8

Conclusion

8.1 Limitations

The way the *total time* was measured when measuring the run-times was far from ideal. What I did in this thesis was taking the total run-time using the `time` command, which made the total run-time include all of the program, including IO operations and checking the results. I have estimated that for the large dataset this typically added an extra 0.5 seconds to the total run-times.

It is not a major issue, one can estimate the time for the entire program without this extra overhead by adding together the times spent on each of the main phases of the program. Additionally, since the most important aspect of this thesis is optimizing the particular methods using OpenACC, for this, the total amount of time spent on the entire program is much less important than the specific run-time of each of the main phases individually.

I experienced some path dependency where after finishing the graphs and various calculations it would be very time consuming to change the the total time where (if I were to do it right) I would also have to rerun all the programs, or I could calculate it from each of the major phases, but that would still require a lot of recalculation of the various percentages. I decided, since it was not all that important, to leave it the way it was.

Another limitation is a possible inefficiency in the way the function "`findNeighborMasks`" was implemented. This function was presented in Section 6.5, and the inefficiency was described in a footnote after the function was presented. The correctness of the program was not impacted by this, though it might have added to the time spent on the "finding potential matches" phase for the single table version of LSH.

8.2 Concluding remarks and result discussion

In this thesis I have touched upon many subjects. I have introduced two versions of the Locality Sensitive Hashing (LSH) algorithm, and I have shown how to program them and optimize them using the GPGPU programming interface OpenACC.

8.2.1 OpenACC

Different aspects of OpenACC programming have been covered. I presented the use of data directives, as well as an example that even when data directives are already present, changing the way the data directives are used can improve the performance of the program.

I showed how one can parallelize certain loops with only semi-independent iterations if the loop behavior is still conventional in specific ways, specifically I showed how, using the OpenACC `reduce` clause, one can parallelize loops that perform common types of reductions.

I briefly discussed one limitation of OpenACC that occurs due to its more simple interface, namely the lack of control over shared memory.

Overall, I was able to see massive speedups using OpenACC. The final program is able to be quickly recompiled for new architectures letting the compiler do most of the code maintenance and architecture specific optimizations. Some tuning of the loop optimization may in some cases be beneficial when running on very different architectures however.

Finally, OpenACC is also incredibly easy to use once you get used to it, what seems like the hardest part is the data movement, but even that becomes relatively simple with only a little practice.

8.2.2 Locality Sensitive Hashing

In Section 5.1 I went through how the program improved for each step of optimization using a standard LSH algorithm that uses multiple tables. These improvements were summarized in Table 5.9 as well as the other tables in Section 5.2. One can see from the results that the use of OpenACC was very successful for optimizing the program, though as I pointed out at the very end of Section 4.3.3 about optimizing the calculation of hash values, there were some limitations about how one might use shared memory, limitations that may or may not have prevented further improvements to the program.

I also presented a version of LSH in Section 6 which, due to the way it is designed, requires only a single table. In general, this version of the algorithm seems to give similar results as the normal version of LSH which uses several tables. This single-table version of the algorithm seems to perform better than the multi-table version overall, though it spends some more time on the actual point matching.

The two versions of LSH were very similar in terms of how they were parallelized using GPUs. The largest differences between the two programs were really about the serial part where potential matches were found. In the multi-table version potential matches were found by searching one box per table. In the single table version however, potential matches were found by checking multiple boxes in a single table by calculating the distances from the query points to the borders of the boxes they get hashed to. Both versions of the algorithm were fairly easy to parallelize, though they were designed to be parallelizable from the start.

Both versions of LSH were implemented from scratch.

8.2.3 Algorithms Compared

When using the two versions of LSH on the large dataset we get similar results. For the multi-table LSH I used 32 LSH tables; then, for each table 16 random hyperplanes were used each represented by a normal vector whose elements sum to zero.

For the single table LSH some improved hyperplanes which were orthogonal to each other and made to partition the search space better were used, each hyperplane was represented by a normal vector whose elements sum to zero.

The performance of the two algorithms was presented in detail in Chapter 5 and 7, with many different tables for many different sets of parameters. Here I will only compare the final results, and give a rough estimate as to how they perform on the large dataset.

The final performance of the multi-table version of LSH is presented by the two first tables in Section 5.1, Table 5.1 and Table 5.2. In short, this version of LSH was able to get a correct ratio of 83.03% in 3.46 seconds, the algorithm searched on average 2.40% of the search space per query point to achieve this. The combined time from simply adding together the time spent on each major phase is only 3.085 seconds however, as I explained in the limitations section this might be more representative of the actual time spent by the algorithm.

Likewise, the final performance of the multi-table version of LSH is presented by the two first tables in Section 7.2, Table 7.1 and Table 7.2. In short, this version of LSH was able to get a correct ratio of 83.36% in 2.44 seconds, the algorithm searched on average 2.73% of the search space per query point to achieve this. The combined time from simply adding together the time spent on each major phase is 1.964 seconds.

We can see that in terms of the time vs. the correct classification ratio, the single table version of LSH wins out despite having to search a larger amount of the search space. The reason for this is that the single table version of the algorithm involves significantly less overhead, both for hashing and for finding potential matches.

It is important to mention however, that these are far from the only versions of LSH out there. I chose these two versions of LSH since I was able to implement them, and since they were very different in nature and interesting to contrast and compare. Using another version of LSH one can probably get better results.

8.2.4 LSH Compared to Exhaustive Search

I will also compare the algorithms to exhaustive search using a normal "naive" CUDA implementation, which simply compares all query points with all base points without any shortcuts; this naive algorithm is then optimized with CUDA without any algorithmic tricks to make the algorithm go faster. This version of the algorithm had a run-time of 15.94 seconds with 100% correct classification. While one can argue that the LSH performs better than this algorithm, this is not the best implementation of point matching using exhaustive search that is out there.

NVIDIA has developed their own algorithm for matching vectors using exhaustive search, which is added to its CUDA Basic Linear Algebra Subroutine library, called cuBLAS. This library contains many linear algebra functions that are heavily optimized for NVIDIA GPUs. One of these algorithms can be used to match many dimensional vectors like feature descriptors.

One can use cuBLAS for input of differently sized datatypes, when using smaller datatypes one can get faster results, but it comes at the cost of some accuracy. Other optimizations made for increased speed also sometimes come at the cost of accuracy. When storing the points using arrays of floats (32 bit), matching using the cuBLAS library has an accuracy of 88.3% finishing in roughly 0.903 seconds. When using halfs (16 bit) the matching goes a bit quicker at the cost of some accuracy, the matching now has an accuracy of 87.7% with a runtime of 0.188s.

When comparing the results from LSH to those from cuBLAS it seems that LSH is less effective than a really well optimized version of the exhaustive search algorithm, at least for the current dataset. But whether LSH really can compete with a well optimized exhaustive search implementation really depends on the use case.

LSH is an approximate nearest neighbor algorithm and it has some different pros and cons. There are several things that can make LSH more effective:

1. The closer the best match is to a query point, the greater the chance is that LSH will find it. If the query point is closer to its best match in the base set, this will increase its chance of finding said best match. When the query point and the best match point are almost exactly the same, a match will be very likely even with few tables for the multi-table version, or a low threshold for the single table version.

2. If LSH does not find a best match, then another close match is likely to be found. The results I have presented in this thesis are predicated on the idea that the absolute best match is required since this is important when the final goal of the application is depth estimation. An application that allows for close matches to be found that are not necessarily the best match will have much more use for LSH than an algorithm that requires the best match where a second best match is not only useless but problematic.

3. The LSH algorithm works better the better and more randomly distributed in space the points are. If the points that should be matched are well distributed, then LSH might provide some better results. This seems like a small point compared to the previous two, but it is worth mentioning.

For the purpose of depth estimation for Alice vision, finding the absolute best match is crucial and anything else is undesirable, this is the main issue that makes LSH ineffective for this use case. Meanwhile, the

SIFT feature descriptors deviate enough from their best matches to not make up for this problem.

There are other use cases for point matching where LSH might be more useful however, such as if you wants to find any instance of an object in a large database and do not care if the match is the absolute best, second best, or the 12th best, as long as it is a good match. For the purpose of depth estimation however, the results of my implementations of LSH, in its current state, are not good enough.

8.3 Future research

8.3.1 Future research regarding Locality Sensitive Hashing

One idea discussed in the thesis that was not implemented yet had some interesting aspects to it was dimensionality reduction before performing a nearest neighbor algorithm such as LSH. Using PCA one can find a matrix that performs dimensionality reduction on data, where the data is projected along its principal directions. One can calculate a matrix that performs dimensionality reduction onto the principal directions of some training set in an attempt to reduce the dimensionality and get rid of redundancy before applying LSH in order to find the nearest neighbors.

If one uses dimensionality reduction, one would of course have to perform the same dimensionality reduction for the query points and the base points, it would also be best to pre-compute this dimensionality reduction for a training set rather than finding the principal directions of the base set since it takes quite a lot of time.

Another idea might be to explore the use of LSH for other applications. As discussed previously, LSH is an Approximate Nearest Neighbor algorithm, both because it is not guaranteed to give the right answer, and because it might give an answer that is close to a nearest neighbor but not the actual nearest neighbor (such as the second or third nearest neighbor). For some applications, a close match is almost as good as the best match, LSH might therefore be more useful in these circumstances.

8.3.2 Future research regarding OpenACC

In this thesis I only covered how OpenACC was used when it was the only GPU programming interface. An area of research that could be further analyzed is the interplay of CUDA and OpenACC, specifically using both CUDA and OpenACC for the same program and in combination with one another.

Another area of research would be to explore how OpenACC can be used for Field Programmable Gate Arrays (FPGA)s. What is possible in this area is still changing as the OpenACC programming model is adapting to these types of problems.

For this thesis the LSH algorithms were programmed in C/C++ using OpenACC directives, compiled using the Portland Group C++ compiler (`pgc++`). The OpenACC programming interface is supposed to work for other compilers however, one may also attempt to use different compilers and compare the results to see how they differ.

This thesis also focused mainly on how to program OpenACC, and how to apply OpenACC In order to optimize LSH. It would be interesting to try to explore how the OpenACC directives together with the serial program actually translates into real lower level code. For the `pgc++` compiler for instance, the `-Minfo` flag enables the compiler to give feedback to the programmer about how the program is optimized, this could be useful if one wanted to examine more carefully how OpenACC directives result in code on the device.

Bibliography

[1] Jeffrey S. Beis and David G. Lowe. "Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces." In: (1997).

[2] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching." In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[3] Pål Halvorsen Carsten Griwodz Lilian Calvet. "PopSift: a faithful SIFT implementation for real-time applications." In: (2018).

[4] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing, Global Edition*. 4th ed. Pearson Education Limited, 2017, pp. 881–898. ISBN: 978-1-292-22304-9.

[5] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. "Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality." In: *Theory of Computing* 8.14 (2012), pp. 321–350. DOI: 10.4086/toc.2012.v008a014. URL: http://www.theoryofcomputing.org/articles/v008a014.

[6] Piotr Indyk and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality." In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 604–613. ISBN: 0897919629. DOI: 10.1145/276698.276876. URL: https://doi.org/10.1145/276698.276876.

[7] Herve Jégou, Matthijs Douze, and Cordelia Schmid. "Product Quantization for Nearest Neighbor Search." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[8] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 3rd ed. Morgan Kaufmann, 2016. ISBN: 978-0-12-811986-0.

[9] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints." In: (2004).

[10] *OpenACC Programming and Best Practices Guide*. openacc-standard.org, 2015.