Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

RNDr. Martin Kruliš

Employing Parallel Architectures in Similarity Search

Departement of Software Engineering

Supervisor of the doctoral thesis: RNDr. Jakub Yaghob, Ph.D.

Study programme: 4I2, Software Systems

Prague, 2013

I would like to give thanks to many people who helped me carry out this thesis. First of all, I would like to thank my advisor RNDr. Jakub Yaghob, Ph.D., who led me kindly through the dangerous realms of parallel programming, who solved many of my technical problems, and who encouraged me in my work.

I would like to thank my family and my fiancée Háňa for their love, patience, and understanding. They have been a great support and they provided a moral courage in the times of need.

Finally, I would like to thank my friends and coworkers for their advice and technical help they gave me on countless occasions. Especially, I give my thanks to Katie for the language and grammar corrections of the thesis, to Zbyněk Falt for the assistance with the experiments on the Bobox system, and to Jakub Lokoč for his insights into the dark arts of the similarity search.

Název práce: Nasazení paralelních architektur v podobnostním vyhledávání

Autor: RNDr. Martin Kruliš

Katedra: Katedra Softwarového Inženýrství

Vedoucí disertační práce: RNDr. Jakub Yaghob, Ph.D.

Abstrakt: Tato práce se zabývá možnostmi nasazení masivně paralelních architektur v databázových systémech využívajících podobnostní vyhledávání. Hlavním předmětem našeho zájmu je využití výpočetní síly současné generace grafických karet pro vyhledávání v databázích obrázků. I přes významný pokrok v posledních letech zůstává oblast podobnostního vyhledávání velmi výpočetně náročná, takže je možné tyto metody aplikovat pouze u databází menšího rozsahu. Grafické čipy disponují obrovskou výpočetní silou, avšak jejich použitelnost pro konkrétní problémy bývá komplikovaná z důvodu specifických vlastností této architektury, které si vyžadují individuální úpravu existujících algoritmů a datových struktur. Zabývali jsme se všemi aspekty této problematiky, od efektivního využití grafických čipů pro obecné výpočty přes akceleraci vyhledávacího procesu až po efektivní indexaci obrázků. Ve většině případů přineslo nasazení grafických karet zrychlení přibližně o dva řády ve srovnání s jednojádrovými procesory a několikanásobné zrychlení ve srovnání s běžnými víceprocesorovými NUMA servery. Tato práce shrnuje naše poznatky z několikaletého výzkumu, algoritmy upravené pro specifické podmínky masivně paralelních čipů, ale také výsledky provedených experimentů, které potvrzují naše závěry.

Klíčová slova: paralelní, databáze, plánování, GPGPU, podobnostní vyhledávání

Title: Employing Parallel Architectures in Similarity Search

Author: RNDr. Martin Kruliš

Department: Departement of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D.

Abstract: This work examines the possibilities of employing highly parallel architectures in database systems, which are based on the similarity search paradigm. The main objective of our research is utilizing the computational power of current GPU devices for similarity search in the databases of images. Despite leaping progress made in the past few years, the similarity search problems remain very expensive from a computational point of view, which limits the scope of their applicability. GPU devices have a tremendous computational power at their disposal; however, the usability of this power for particular problems is often complicated due to the specific properties of this architecture. Therefore, the existing algorithms and data structures require extensive modifications if they are to be adapted for the GPUs. We have addressed all the aspects of this domain, such as efficient utilization of the GPU hardware for generic computations, parallelization of similarity search process, and acceleration of image indexing techniques. In most cases, employing the GPU devices brought a speedup of two orders of magnitude with respect to single-core CPUs and approximately one order of magnitude with respect to multiprocessor NUMA servers. This thesis summarizes our experience and discoveries from several years of research, related algorithms adopted for the specific conditions of GPU architectures, and the results of empirical experiments performed in order to verify our claims.

Keywords: parallel, databases, scheduling, GPGPU, similarity search

Contents

1. Introduction

Software efficiency has been an intensively discussed issue since the very dawn of the computer science. Efficiency has been affecting many attributes of information systems, such as response times or the maximal size of the data we can process in a reasonable time. In the past, the performance of an application was determined mostly by the selection of appropriate algorithms and data structures, quality of the implementation, the compiler used, and the computational speed of the processor. The instruction processing speed was the leading factor since major improvements of most algorithms are quite rare and the optimal implementation is usually tailored to the compiler and the CPU architecture.

The computational speed of a processor was tightly linked to its operating frequency and the frequency corelated strongly with the number (and size) of the transistors on the chip. As the number of transistors has doubled every one or two years [1, 2], the performance of computers, and thus the software, was leaping steadily. Unfortunately, this trend reached an impasse at the beginning of the 21st century. It was discovered that the heat production of silicon-based chips is cubically proportional[1] to the frequency of the chip. Even though the cooling technologies have developed rapidly to match the needs of the CPUs, the frequency-boosting approach become unsustainable in the long run.

The frequency pursuit was abandoned and the CPU development ventured into the domain of multi-core parallelism. Current mainstream processors are equipped with multiple computational cores which are quite independent. They share only the die casing and a few resources such as the external busses, the memory controllers, or the L3 cache. Most of the Intel CPUs also employ the hyper-threading technology, which maps two virtual CPU cores to one physical, so the internal (and often redundant) units of the core can be better utilized. Furthermore, we can observe that the nonuniform memory architecture systems (NUMA), that encompass multiple physical processors (each managing its own part of the memory), are becoming increasingly popular. The combination of these factors has caused every up-to-date server to have tens of CPU cores, which all need to be utilized in order to achieve an optimal performance.

Another major hardware revolution happened in the field of graphical processing units. In 2006, the GPU stream processor architectures evolved to a point where the GPUs were capable of processing general computational tasks in addition to the traditional image operations and 3D graphic rendering. This generation of generic purpose GPUs gave us a highly parallel architecture capable of processing data at speeds that cannot be achieved even by the best multi-core CPUs. Graphical chips of the day contain from hundreds to thousands computational cores. Unfortunately, this architecture is bound with many limitations concerning the parallel execution model and the memory transfers, which restricts its applicability to rather specific data-parallel problems.

In addition to multi-core CPUs and many-core GPUs, other parallel platforms available for common PCs and servers have been introduced recently. For

[1]Actually, the $P = CV^2f$, where P is power, C is capacitance, V is voltage, and f is frequency [3]. According to frequency-voltage configuration tables, the V is approximately lineary dependent on f.

instance, the IBM Cell processors [4] or the Intel Many Integrated Core (MIC) architecture [5]. These platforms only prove that the trend of parallelization is quite strong in the current hardware and that we need to adapt our algorithms, implementation techniques, and programming paradigms in order to fully exploit their computational power.

1.1 Embracing Parallelism

It has been established that parallelism is one of the most essential things affecting the efficiency of current applications and there are many computational problems that can really benefit from a concurrent execution [6, 7, 8]. We would like to discuss a few very important issues that rise with the introduction of parallelism into algorithms before advancing further.

Performance Evaluation

As our work focuses on improving the performance of applications by embracing parallelism, we need to address the issue of performance evaluation. The theoretical approach, which operates with well established time complexities of algorithms, is not quite satisfactory in this case. On the other hand, the naïve approach of measuring the real execution time is highly dependent on various hardware factors and it suffers from significant errors of measurement. Unfortunately, the real running time is the only practical thing we can measure with acceptable relevance. In the light of these facts, we will provide most of the results as the *parallel speedup*, which is computed as

$$\text{Speedup} = \frac{t_{serial}}{t_{parallel}}$$

where t_{serial} is the real time required by the best serial version of the algorithm and $t_{parallel}$ is the real time taken by the parallel version. Both versions are executed on the same data, thus solving exactly the same problem.

The speedup is always provided along with the number of cores (threads, devices, ...) used for the parallel version of the algorithm. We are trying to reduce the error of measurement by timing the serial and the parallel version of the algorithm on the same machine, using the same compiler, and under the same conditions. If we provide real times in our results, they should be perceived only as illustrational and the emphasis is on the speedup measured.

Scalability and Amdahl's Law

We usually measure the speedup in several different settings, when the parallel implementation utilizes a different number of cores. These tests are designed to asses the *scalability* of the algorithm. In other words, how many computational units can be efficiently utilized, or how well is the problem parallelable. In an optimal case, the speedup is equal to the number of computational units used (i.e., $2\times$ on dual-core, $4\times$ on quad-core, etc.) and we denote this case the *linear speedup*. The scalability also helps us predict how the application will perform in

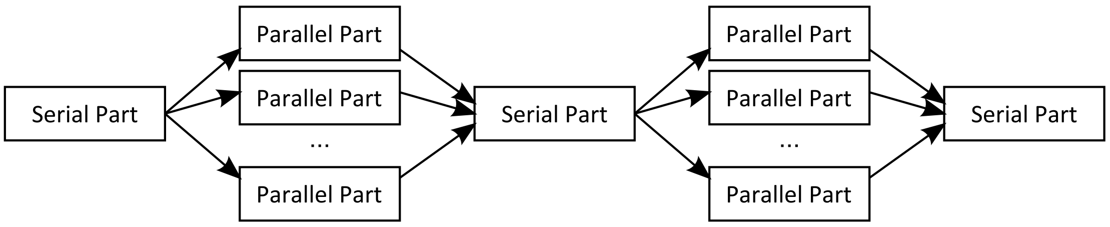


Figure 1.1: An example of the algorithm decomposition

the future as each new generation of CPUs and GPUs has more cores than the previous generation.

The scalability of an algorithm can also be determined by measuring the ratio of its serial and parallel parts (as depicted in Figure 1.1). If we identify the sizes of these parts, we can use the Amdahl's Law [9]

$$S_N = \frac{1}{(1-P) + \frac{P}{N}}$$

to estimate the speedup in advance. The S_N denotes speedup of the algorithm when N computational units are used and the P is the relative size of the parallel part of the algorithm. The speedup estimation becomes particularly interesting when the N tends to the infinity:

$$\lim_{N\to\infty} S_N = \lim_{N\to\infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{1-P}$$

Even though this might sound excessively theoretic, it often helps us understand, what happens when an algorithm is moved from multi-core CPUs with tens of cores to a multi-GPU system with thousands of cores. For instance, if the serial part of the algorithm takes 5% of total work, we will never be able to achieve greater speedup than $20\times$, no matter how many cores we can employ. In such case, we can observe $3.48\times$ speedup on a quad-core CPU (which looks adequate); however, we may achieve only $19.3\times$ speedup on a 512-core GPU card[2]. Therefore, one of our main objectives is to reduce the serial parts as much as possible even at the cost of using an algorithm with suboptimal time complexity.

1.2 Outlining Objectives

This thesis focuses mainly on the problematics of database systems that employ similarity search and content-based retrieval. The main objective is to identify the key points that will benefit the most from parallelism and to exploit them to improve the performance of these systems. A particular emphasis was put on the utilization of generic purpose GPUs as they were the only highly parallel platform generally available at the time our research begun.

Current hardware architectures are quite complex. Understanding the design of these architectures is essential for developing an optimal parallel algorithms. Therefore, we dedicated Chapter 2 to revise the multi-core CPU and many-core GPU architectures of the day.

[2]This example is only illustrational as we inadequately compare results of two platforms.

The first part of our research addresses the problems introduced by using a heterogeneous computational platform, where the operating systems and the main part of the application runs on a multi-core CPU(s), but computationally expensive parts are accelerated by the GPU devices. The task scheduling becomes more difficult as the GPU may wait for the CPU or vice versa. We have studied related problems and propose some solutions in Chapter 3.

The second and the most important part of the work is dedicated to the various problems of multimedia databases, especially the efficiency of content-based retrieval and similarity search in large image databases. We have identified several algorithms that would benefit greatly from parallelism. These algorithms and their parallel modifications are described in Chapter 4.

Finally, we focus on the problem of feature extraction, which is used to create the image descriptors for the similarity search. Our main objective is to accelerate the extraction process, so we can index larger datasets in a feasible time and perform many experiments to tune the configuration parameters of the extractor. Our results are summarized in Chapter 5.

1.3 Contributions

This work summarizes the results of three years of individual research on the given topic. The contributions can be identified as follows:

- We have revised the topic of task scheduling in frameworks for parallel data processing and made two improvements [10]. First, we have proposed a new approach for dealing with blocking tasks in parallel frameworks for multi-core CPUs. Second, we have designed our own framework for hybrid CPU-GPU systems. This framework is built on top of the OpenCL library and it simplifies the design of our applications. Furthermore, the proposed idea of the feeding thread pool exhibits significant improvement. It reduced the waiting times during data transfers and allowed us to better utilize the GPU computational power, especially in multi-GPU configurations.

- The domain of similarity search and content-based retrieval is much more computationally demanding, so it can really benefit from the parallel approach. We have adapted a Signature Quadratic Form Distance function, that computes (dis)similarity of two image signatures, for the GPGPU platform [11, 12]. A speedup of two orders of magnitude was observed. Furthermore, we have also accelerated the database access method called the pivot table prefiltering and proposed a novel range estimation algorithm that solves the problem of parallel k nearest neighbour queries.

- Finally, we have employed the GPGPU in the database indexing and proposed a GPGPU feature extractor, which computes the signatures for images [13]. This fast extractor opened new possibilities as it allowed us to index large image databases. It also allowed us to explore the configuration parameter space of the extractor and to find a configuration that produces the most accurate signatures. These experiments took only several weeks on the GPU, but they would have taken more than a year using only CPUs.

2. Parallel Architectures

Knowing your enemy is the first step in designing a good strategy. To do so, we revise current parallel architectures especially multi-core CPUs and many-core GPUs. As these architectures are quite complex, we narrow our focus only to the aspects that directly influence the design of our parallel algorithms.

2.1 Multi-Core CPUs

Multi-core CPUs are well established in the segments of servers, personal computers, and laptops. At present time, they are penetrating to the segment of tablets and mobile phones. Desktop CPUs have up to 16 cores and more cores per chip are expected in future generations. Even though there are many types of processor architectures, we will focus on the Intel IA32 (more commonly known as the x86) architecture, since it is the predominant architecture in personal computers, small servers, and server clusters. Most of the observations made about IA32 hold for other architectures as well and we explicitly point out any important differences.

2.1.1 Parallelism in CPUs

Current CPUs employ parallelism on three levels:

- instruction execution,
- vector instructions (data parallelism – SIMD),
- and on-chip core replication (task parallelism).

Executing Instructions Simultaneously

The very first computers were designed to execute one instruction at a time. Quite soon, it become clear that this solution is suboptimal as each instruction comprises several smaller steps, such as decoding or activation of some numeric processing unit. A natural solution to this problem is to create an *instruction pipeline*. The pipeline design was first mentioned in the work of Konrad Zuse [14] as it was used in the Z1 and Z3 machines. It was employed on regular scale in the late 1970s, especially in the Cray supercomputers [15].

Each instruction is divided into fixed number of steps that are processed by different stages of the pipeline. Separate stages can perform their tasks concurrently, thus the execution of subsequent instructions partially overlap. A simple example of a 4-stage pipeline is depicted in Figure 2.1.

Although the pipeline can increase the instruction throughput significantly, it is encumbered by several problems. Most important are *instruction dependency hazards* and *branching problem.* The first problem is caused by the sequential execution model that is presented to the programmer but not upheld by the pipeline architecture. When instruction inputs depend on the outputs of the previous instruction, the previous instruction needs to be processed entirely (i.e., including

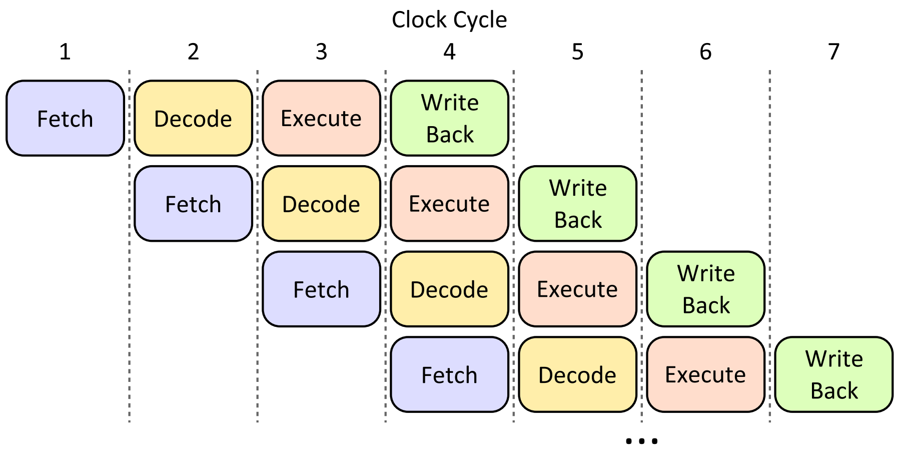


Figure 2.1: An illustrative example of 4-stage instruction pipeline

the write back stage), before the following instruction is executed. Current CPUs deal with this problem by stalling the pipeline or reordering the instructions. The second problem is caused by the uncertainty of conditional jumps. Each conditional jump has two possible results – it is either performed or it is not. This means that the following instruction is not certain until the jump instruction is executed. This problem is usually solved by *branch prediction* and *speculative execution*. The CPU tries to guess, which branch is going to be taken and starts to execute it. If the other branch is taken, the pipeline must be discarded and then repopulated by the correct instruction stream.

The *superscalar architecture* presents the next level in the instruction parallelism. This architecture was first introduced in CDC 6600 Cray mainframe in 1965 and the first x86 CPU with superscalar architecture was Intel Pentium (P5), released in March 1993. A superscalar processor is capable of executing multiple instructions at once. It is usually based on the pipeline architecture, hence a superscalar CPU has more than one pipeline.

Current CPUs also employ the *out-of-order execution*. The x86 processor family first introduced this feature in Intel Pentium Pro (1995) along with the speculative execution. The out-of-order pipeline can reorder instructions to better utilize computational units of the CPUs and avoid instruction dependencies in the pipeline while maintaining the consistency of the results.

Even though this type of parallelism is very interesting and more advanced optimizations are implemented with every new generation of CPUs, it can be hardly affected by the programmer. The instructions may be generated so that the pipeline and the out-of-order execution works better in some cases; however, these optimizations are performed solely by the compiler.

Vector Instructions and Data Parallelism

The single instruction multiple data (SIMD) execution model is based on a simple idea that the processor can perform the same instruction on multiple data simultaneously. This approach is quite conservative and easy to implement as it requires that the processor design duplicates only the execution units (e.g., arithmetic units) and remaining units, such as the instruction decoder, caches, registry, or the data loading/storing units, are shared. Therefore, a vector instruction is treated almost as a regular instruction, except for the execution part which activates multiple symmetric units at once and lets them simultaneously process a vector of numbers instead of a single number.

The x86 family first introduced the MMX vector instruction set in Intel Pentium MMX (1997). It was shortly followed by the 3DNow! technology from AMD (1998) and the Streaming SIMD Extension instruction set (SSE) in 1999. Current x86 processors implement SSE version 4.2, Advanced Encryption Standard (AES), Advanced Vector Extensions (AVX), CVT16, and eXtended Operations (XOP) instruction sets. Most recent instruction sets of fused multiply-add operations (FMA3 and FMA4) have just appeared in the newest AMD processors and are planned for the next generation of the Intel processors. It is safe to say that this level of parallelism has strong support from the CPU vendors, thus we have to embrace it in our software designs.

Fortunately, current compilers are well aware of the vector instruction sets and they try to generate these instructions whenever possible. In complex situations, when the compiler fails to recognize an opportunity to exploit vector instructions, we can optimize critical routines manually by rewriting them in assembly language, or by using some special libraries, such as the `xmmintrin.h` header, which provides API for vector instructions directly from C/C++ language.

Embedding Multiple Cores for Task Parallelism

The third level of parallelism is the one that concerns us the most. As the transistors are getting smaller with each new generation of chips, it become possible to integrate multiple CPU cores onto a chip (as depicted in Figure 2.2). These cores are almost as independent as separate processors would be, since they share only communication buses, power management, and top level L3 cache. Each processor core executes separate instruction thread, thus they are perfectly suited for task-parallel problems.

Furthermore, a CPU core has many redundant units. Despite the instruction level parallelism, these units are rarely all occupied. One of the possible solutions is to increase the number of threads processed by the core. It can be achieved by attaching multiple frontends (*logical cores*) to each physical core. The logical cores appear as regular processors to the operating system, but they share many resources of the physical core. The instructions issued to these cores are interleaved in some way, hence they increase the utilization of internal units and mask some of the system latencies.

One of the first systems that implemented this approach was the Delencor HEP [16] in early 1980s. It used multiple blocking threads per core interleaved on cycle-by-cycle basis in a pipeline. Currently, the multithreading technique is implemented in various architectures, such as Itanium (IA-64), IBM POWER5, or UltraSPARC.

The first appearance in the x86 family was in 2002, when Intel introduced new Pentium 4 Netburst architecture [17] with *Hyper-threading* technology [18, 19]. Each core has two logical frontends, thus it processes two independent instruction streams. Intel chose a minimalist approach as they added only the most essential units like registry alias tables or instruction buffers to the chip. Most of the remaining units are either time-shared or partitioned between the two threads.

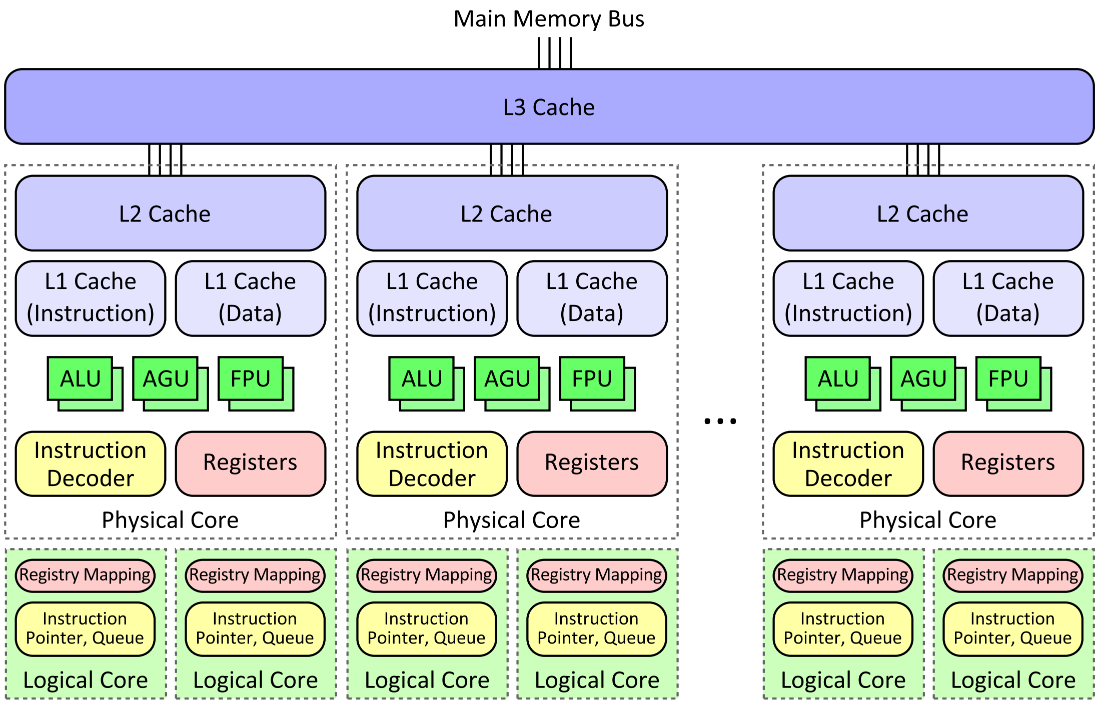


Figure 2.2: An illustrative schema of a multi-core CPU

AMD implemented their version of multithreading almost ten years later in the Bulldozer architecture [20]. Unlike Intel, the AMD *dual-core modules* actually duplicate many of the units. The idea is to create two cores with much simpler design but dedicated units, especially the L1 cache and the integer execution units. However, some of the parts (like L2 cache, instruction decoder, or floating point units) are still shared between the cores.

As we have mentioned in Chapter 1, the multi-core approach was an answer to the heat problems caused by high operating frequencies. Even though the tradeoff between the number of cores and their frequencies is quite beneficial for large servers and multiprocess environments, there are still applications, which are inherently serial. In order to increase performance of serial problems, most of the current CPUs implement some kind of internal speed regulation (e.g., the Intel Turbo Boost technology). In case some cores are not utilized, the CPU power management turns these cores off and diverts the saved energy to the remaining cores, so they can increase their frequencies as they got more power. This technology must be considered as in some cases it might be better to use a fast serial algorithm instead of an inefficient parallel one.

2.1.2 Memory Issues

Most of the tightly coupled parallel systems rely on *shared memory* model, where multiple computational units share their data in one memory space. This model has several important issues that must be addressed, like the synchronization, coherency, performance, or memory protection. We revise issues which are the most important for the design of parallel algorithms.

Data Synchronization

In any parallel environment, access to shared assets has to be synchronized. In case of shared memory, access to mutable data structures shared amongst multiple threads must be coordinated to avoid data corruption caused by concurrent read-write and write-write operations. Even though there are many techniques and design patterns [21] that reduce the necessity of synchronization, like data replication or privatization, the synchronization is inevitable in many situations.

The problematics of synchronization has been thoroughly studied [22, 23, 24]. There are basically two ways, how to synchronize access to shared data:

- atomic operations and
- mutual exclusion.

The *atomic operations* are suitable for simple stand-alone updates of the data. An atomic operation is a single instruction that is guaranteed by the system architecture to be performed entirely at once and in a thread-safe manner [25]. The most common operation is the *compare-and-swap* instruction (CAS), also known as compare-and-exchange or test-and-set. The instruction has three operands: old value, new value, and a pointer to variable in main memory. If the variable holds the old value, the new value is atomically assigned to it. Even though we can implement almost anything with this instruction, specialized atomic instructions such as increment, integer arithmetics, or logical functions are implemented by most architectures [25].

If data updates are more complex and the functionality cannot be provided by atomic operations, the *mutual exclusion* needs to be ensured. It is based on the premise, that only one thread may work with the shared data at a time. There are many types of synchronization primitives. The simpler ones use guarding variable (a lock) which is modified by atomic operations (usually the CAS instruction). An active waiting is used in case the lock is acquired by a different thread. A typical representant of such primitive is the *spin-lock*.

More complex primitives like the *mutex*, the *semaphore*, or the *read-write lock* suspend the waiting thread to save computational resources. In order to do so, in addition to guarding variable, they require queues for waiting threads and a mechanism for waking suspended threads.

Caches

Memory latency is a serious performance problem since the CPU processing speed significantly outmatches the data throughput of the operating memory. There are various techniques that can be used to reduce memory latency. Common CPUs deal with the problem by employing multi-level caches that selectively store fragments of data which are used by the CPU core at a time (see Figure 2.2). Current x86 CPUs use three-level organization. L1 caches, which are closest to the CPU core, are the smallest (tens of kB) but also the fastest. L3 caches, which are closest to the main memory, are the largest (several MB) and the slowest.

The lower levels of cache (usually L1 and L2) are replicated on every core, while higher levels (L3, sometimes L2) are shared. This presents a potential problem as the same data might be copied and modified in caches of two cores independently. In order to maintain data integrity, some kind of cache coherency protocol must be employed. The x86 architecture use the *MESI protocol* [25] in combination with *memory bus snooping* technique.

The protocol marks every cache line with label *Modified*, *Exclusive*, *Shared*, or *Invalid*. These labels indicate the replication state of the data. There are simple rules describing which operation is the processor core allowed to do with each line type. The protocol also defines how to change the state of the line if necessary. The processor is snooping on the memory bus and modifies the state of the lines according to the observed memory traffic. This coherency protocol is operating on each level of the cache.

A cache coherency protocol can create an unpleasant side effect called *false sharing*. Since a cache line is usually at least tens of bytes long (64 B in IA32), two threads might be working on independent data which are close enough to fall in the same cache line. In such case, the coherency protocol forces the cores to steal the cache line every time they would like to perform some exclusive operation, thus creating a ping-pong effect on the memory bus. This problem is easily avoided by aligning intensively used data to the cache line boundary.

Even though we have focused on the x86 CPU family, most of other architectures use very similar mechanisms including the cache coherency protocols. These differences are not important from the programming point of view, thus we have chosen not to elaborate on the details.

Virtual Memory Space

The memory protection mechanisms used by current operating systems [22, 23] to separate data of different processes introduce virtual memory spaces. Each process is provided with its own virtual memory space, which creates an illusion the process has all the memory for itself. It also prevents the process to see or write the memory of other processes, unless two processes explicitly negotiate some memory sharing.

Virtual memory addresses need to be translated to physical memory addresses. This translation mechanism must be supported directly by the CPU and the operating system must manage the translation mechanism. The IA32 architecture uses page tables to translate the address [25]. Page tables are organized as a widespread shallow tree[1], which is quickly traversed from root to leaf when a physical address is looked up. Table on each level is indexed directly by a part of the virtual address, thus the *page-walk* is very fast. On the other hand, these tables are present in main memory, thus each translation requires several memory reads.

The main reason, why the virtual address translation is not incredibly slow, is the presence of a dedicated cache for translated addresses called *translation lookaside buffers* (TLB). In order to utilize this cache as much as possible, we should avoid data access patterns that jump over large portions of the virtual memory space. Furthermore, the TLB is cleared every time a process is switched on the CPU core, since the new process has its own memory space. Hence, we should minimize context switches and use threads that share memory space rather than processes with their own memory spaces.

[1]With up to 4 levels, depending on address space type and size.

Other architectures use more or less elaborate translation mechanisms. One of them is the *inverted page tables* translation employed by PowerPC, UltraSPARC, or IA-64. The basic idea is to use hashing in order to find translation for an address quickly. Another approach is to use a TLB-only solution with software filling like MIPS processors does. The TLB cache is manipulated by the operating system and cache misses are handled by system exceptions.

NUMA Systems

Small symmetric multiprocessor systems are often organized in cache coherent nonuniform memory architecture (ccNUMA) [26]. In this architecture, each (multi-core) processor has its own private memory and it is interconnected with one, two, or even three other processors as depicted in Figure 2.3.

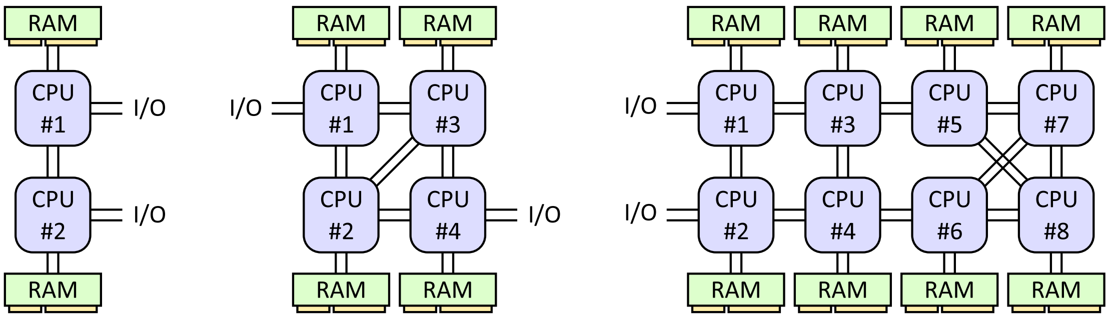


Figure 2.3: Samples of 2, 4, and 8 node NUMA systems

Each node can access not only its own memory, but the memories of other nodes as well. Obviously, data transfers from the local memory of the node are faster than transfers from a memory of another processor. This delay is often called the *NUMA factor* and it can slow the data transfers even several times. Furthermore, it has been observed that when a memory of a processor is accessed by another processor, the first processor is being slowed by the data transfer.

2.1.3 API and Parallel Libraries

Parallel execution on multiple cores is achieved by employing multiple processes or by spawning multiple threads within one process. The technical and implementation details are within the realms of the operating system kernel, which is well beyond the scope of this thesis. As we are focusing on high performance data processing applications, we will settle for the simple fact that the underlying operating system allows us to execute multiple threads at once and the threads are executed by available processing units in the best effort manner. Each thread is a piece of code that may run independently (i.e., it has its own state, call stack, etc.), but all threads share the memory, thus they can easily cooperate on a problem.

The threads provide a low-level generic way how to execute code in parallel on multi-core or multi-CPU systems, but they are not very programmer-friendly. To reduce the tedious programmers' work, various parallel libraries, frameworks and application interfaces have been introduced, such as the Intel Threading Building Blocks [27, 28] or the OpenMP [29, 30]. These libraries provide extensions to standard thread programming by offering additional value, such as parallel algorithm templates or data structures designed for concurrent access. In the following, we briefly describe the Intel TBB and some of their generic ideas, as we have used this library in our experiments.

Intel Threading Building Blocks

The Intel TBB is an open source multi-platform library for the standard C++ language. Unlike some other libraries it does not provide any language extensions, but it uses C++ templates and other C++ features to provide generic parallel algorithms and data structures in addition to standard things like threads or atomic operations. We introduce some of the most essential structures to illuminate the general idea. More detailed descriptions can be found in Intel documentation [28] and literature [27].

Basic parallel algorithms, which are sometimes called *parallel primitives*, cover the most fundamental problems.

- The *parallel-for* is a template for a typical data parallel task. It traverses a selected index range and invokes a given functor (the body of the parallel loop) for each item in the range (concurrently when possible).
- The *parallel reduce* performs standard tree reduction technique, where leaf operations (computation) as well as joining operations (reduction) are performed in parallel.
- Finally, the *parallel scan* performs the prefix scan operation concurrently. The prefix scan takes a vector x_i and computes another vector y_i, where each $y_i = x_i \oplus y_{i-1}$ ($y_0 = x_0 \oplus Id_\oplus$). The $\oplus$ is an associative operation and $Id_\oplus$ is a neutral item of the $\oplus$. Even though the prefix scan looks inherently serial, it can be parallelized at the cost of doing some additional work [27].

Parallel data structures implemented in TBB are basically a thread safe versions of some C++ STL containers. For instance the TBB *concurrent vector* or the *concurrent queue* are very similar to the *vector* and the *dequeue* STL containers, but they can be operated simultaneously from multiple threads or parallel primitives. Others, like the *concurrent hash map*, are specifically designed for concurrent usage and provide a specific API that is both effective and efficient.

Task Scheduling

The TBB offers a quite sophisticated task scheduler besides the convenient parallel primitives and containers. This scheduler also works as an engine for the primitives described above. A task scheduler is more convenient than programming with raw threads as it reduces the overhead of creating and disposing of a thread and provides better ways to keep the CPU workload balanced.

The scheduler uses a *thread pool* – a pool of worker threads which are created when the application starts (or when first needed) and destroyed when the application terminates. These threads are waiting on a synchronization primitive until a task is dispatched to them. Waking up a suspended thread is much faster than creating a new one, thus a significant amount of time can be saved. The

TBB usually creates as many threads as there are logical cores available. This way, all cores can be occupied if sufficient concurrent tasks are available and the overhead of the operating system scheduler is reduced as the threads do not need to switch between available processors.

The tasks are dispatched nonpreemptively to free threads. If no free thread is available, they are queued and dispatched as soon as one of the worker threads finishes. Since the dispatching overhead is quite low, the programmers are encouraged to produce large quantity of tasks. This technique is called *oversubscription* and it helps to balance the workload. We address this matter more thoroughly in Section 3.2.2.

2.2 Many-Core GPUs

A graphical processing unit is a piece of specialized hardware originally designed to encode digital image data into signals that can be directly interpreted by the computer monitor. First GPU cards comprised mostly the video memory connected to digital-analog signal converters, and their programming API provided only very simple 2D operations for copying image data. In 1996, the first 3D accelerator for desktop PCs was presented and the GPU encompassed some basic computational operations required for rendering 3D graphics, such as fast multiplication of small vectors and matrices. Propelled by the gaming industry, the GPU development raced forward and featured new functions like textures rendering or lightning computations.

In 2001, the 3D rendering pipeline was enhanced by introducing small pieces of code called shaders, that can be executed over every vertex of the scene or over every pixel fragment in a highly parallel fashion. The stream processors that execute shaders were quite primitive in the very first version and their program was limited in length, instruction set, and available memory (registers). In the following five years, the stream processors become much more powerful and universal. Finally, in 2006, they have become powerful enough to be used for generic (i.e., not only graphical) computations as well. The GPGPU has emerged as a very strong and cheap parallel platform designed for data parallel problems. As such, it plays and is going to play a significant role in high performance computations.

In this section, we summarize the hardware properties and programming model of current GPUs. Most of the facts and observations are made about NVIDIA Fermi architecture [31], which was the state of the art when our experiments begun. The newest state of the art architecture (NVIDIA Kepler [32]), which emerged in 2012, features some enhancements. However, we do not provide a description of Kepler architecture as all our experiments were conducted on the Fermi GPU cards. At the end of the section, we also provide a brief description of the programming API – the OpenCL library, which was used in our prototype implementations.

The development of GPUs lies solely in the hands of commercial companies, which keep their secrets about implementation details quite safe. We can only summarize information published in programming specifications [33], optimization guidelines [34], and related literature [35], hence our overview might be slightly inaccurate. We describe the hardware model as it appears to the programmer rather than the hardware itself.

2.2.1 Hardware Overview

A GPU card is an independent device (see Figure 2.4) with all advantages and disadvantages. The main advantage is that the device can process its own tasks, while other parts of the system, especially the CPU and other GPUs, can perform other work. The main disadvantage is that the GPU card is separate from the host operating memory, hence all input data must be transferred from the host to the GPU internal memory and all results must be transferred back. Furthermore, the CPU cannot interrupt nor interfere with the GPU tasks once they have been dispatched.

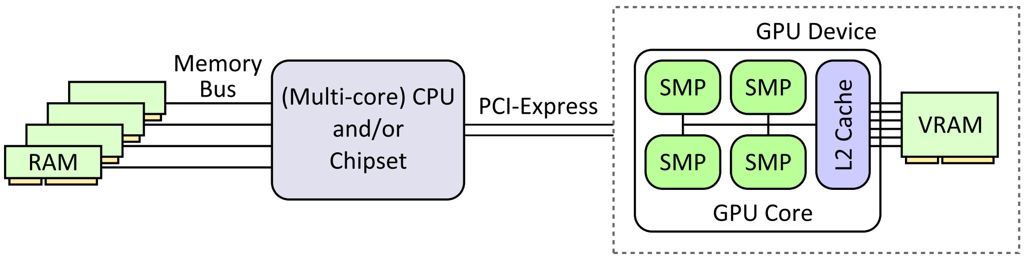


Figure 2.4: Schema of a host system with a GPU device

The GPU is interconnected with the host system by an expansion bus, usually the PCI-Express (PCIe). The PCI-Express 2.0 (16×), which was present in our hardware, is capable of transferring up to 8 GB per second in both directions. In comparison with CPU buses, such as Intel QuickPath Interconnect (QPI) or AMD HyperTransport (HT), which both have throughput of approximately 25 GB/s, the PCIe bus is rather slow. Therefore, the data transfer cost must be considered carefully and the GPU should be used only for operations, where the amount of the computational work significantly overweights the data transfers, or when the data transfers can overlap with the computational work.

The device itself is equipped with an internal memory and a GPU processor. They are connected by a wide memory bus which has high throughput – usually over 100 GB/s. Beside the shared memory controller and the L2 cache, the GPU processor is formed by several *Symmetric Multiprocessors* (SMPs). A fully loaded Fermi GPU carries 16 SMPs.

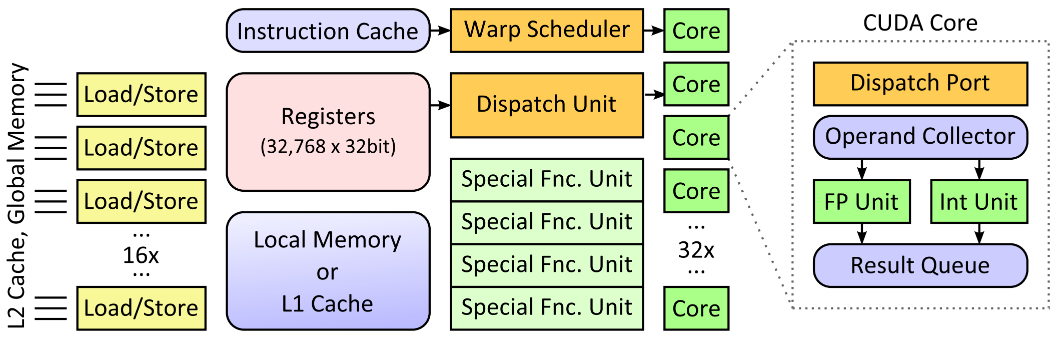


Figure 2.5: Schema of GPU symmetric multiprocessor

The SMP itself (Figure 2.5) consist of several computational cores (32 in case of Fermi). Each core has its own arithmetic units for integer and floating point math and a private set of registers, but they share some resources of the SMP.

Most importantly, the instruction cache, the warp scheduler, and the dispatch unit. These are responsible for the thread planning and the instruction execution, thus all cores must execute the same instruction at the same time. This model is discussed in more detail in the following section (2.2.2). The cores also share load/store units, that are responsible for main memory data transactions, the L1 cache, and the *local memory.* We explain the details of memory structure later, in Section 2.2.3.

2.2.2 Execution Model

The GPU devices embrace the data parallel model. In this model, one function (called *kernel*), is invoked multiple times for multiple inputs. The parallelism is achieved by processing these inputs concurrently. This model works perfectly in cases when the kernel function operates solely on the input objects without accessing any other data, thus completely without synchronization. Furthermore, there should be enough data objects to occupy all available processing units to fully utilize the hardware.

When a kernel is invoked, the programmer defines, how many threads it will spawn. All these threads execute the same code with the same arguments, but they are also provided with a *thread ID* value, which can be used to determine the portion of the work (i.e., the input data item) the thread should process. The ID is usually a flat number (from 0 to $N - 1$ where N is the number of threads spawned), but it can also be a two or three dimensional vector, so the programmer can conveniently navigate in 2D and 3D problems. The threads are lightweight entities as they share almost everything, except for the ID. The programmer is encouraged to create a huge number of threads as the system can easily encompass millions of them. A larger number of threads with smaller tasks to perform creates more balanced workload for the GPU cores.

Threads with adjacent IDs are bundled together in groups. More precisely, the programmer specify the group size G when invoking a kernel[2] and the t_{ID} thread is placed in the $\lfloor t_{ID}/G \rfloor$ group. The groups are important from the perspective of scheduling as well as from their perspective of sharing resources, as we will cover later.

Single Instruction Multiple Threads

This execution model is an extension of the Single Instruction Multiple Data (SIMD) parallelism which is employed in the CPU vector instructions such as SSE. In this case, we have multiple threads that all have the same code, but they advance through the code together executing the same instruction at a time. However, each thread has its own set of registers, thus working on different data.

This concept is much more powerful than SIMD as the threads have their own memory and even though they must execute the same instruction at a time, they can use branches or even while-loops. Furthermore, the code written for the SIMT model is more clear than a code which contains vector instructions. Finally, the work synchronization amongst the threads is trivial and the barrier instruction is in fact only a simple memory fence.

[2]In case of multidimensional IDs, the group size is defined for each dimension.

On the other hand, SIMT execution can be used efficiently only in case all threads have exactly the same amount of work. The main problem is branching. When different threads execute different branches of an if-statement, all cores must execute both branches and some kind of instruction masking technique must be employed, in order to render instructions in the invalid branches inactive. In the worst case scenario, the threads execute a while-loop that terminates early for most of the data, but it can run for a long time in a few isolated cases. If so, most of the cores will linger needlessly in the while-loop masking their instructions while only a few cores will be doing any real work finishing their long-running tasks.

Threads on The Symmetric Multiprocessor

Thread scheduler plans thread groups to available SMPs, usually one group to one SMP. Theoretically, there might be multiple groups executing the same kernel mapped to a SMP, if the groups have low requirements and the SMP resources, such as registers or local memory, are underutilized. A thread group is never split amongst multiple SMPs as the threads in the group require access to the local memory.

The group may have more threads than there are available cores on the SMP, so the threads are divided into *warps*[3]. A warp is a kind of a subgroup with size equal to the number of cores in the SMP. Only one warp is actually running, while other warps are waiting to be scheduled. Threads in one warp are implementing the real SIMT model, as the SMP cores are running in *lock step*, and the threads in one group are running in virtual SIMT. The context switch between two warps is very fast, so the scheduler changes them very often, especially if the running warp becomes stalled (by a memory transaction, for instance).

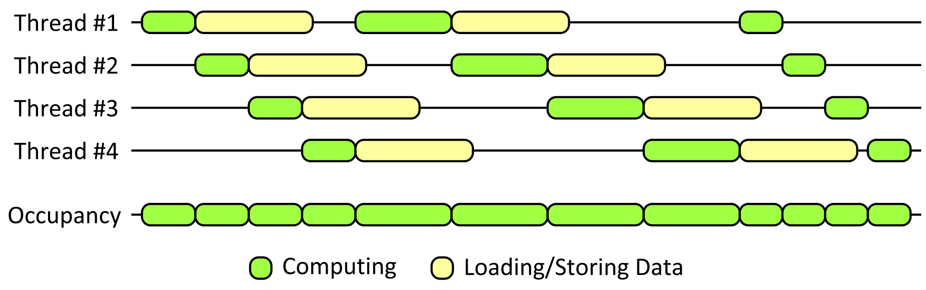


Figure 2.6: Thread overlapping, hiding memory latency

Figure 2.6 depicts the principle of reducing memory latencies by overlapping memory transfers with computations. For instance, when the running warp executes an instruction that loads data from global memory, the load/store units start the memory transaction and the threads must wait for the data. This might be a lengthy operation, so the scheduler switches to another warp which can run meanwhile. The warps are scheduled in a round-robin fashion, thus when the first warp is scheduled again, it is quite likely the data transaction has finished and the warp can continue immediately.

At this point, we need to emphasise an important issue of the thread scheduling. The thread group is assigned to a SMP nonpreemptively. It means that

[3]AMD denotes them *wavefronts*.

all threads must terminate their work before another group is scheduled to that SMP. The context switch of two groups on a SMP would be quite expensive and unnecessary for most data parallel algorithms. On the other hand, there is no way of creating a task synchronization primitive (e.g., a barrier) over all groups, unless we can guarantee that all the groups are assigned to the SMPs and running. Otherwise, any such primitive would most certainly cause a deadlock of the whole system.

2.2.3 Memory Structure

The GPU has much more complicated memory model than the CPU. A CPU process perceives the memory as a single uniform space with linear addressing. The GPU process has to deal with several types of memories with different address spaces. Beside the host memory, which is not directly accessible from the kernel, there are following memory spaces:

- a global memory,
- a constant memory,
- a local (shared) memory,
- and a private memory.

Each memory space has separate addressing, thus different type of pointers. The kernel must explicitly declare the type of the address space when creating a pointer and pointers of different address spaces are not compatible.

The memory spaces differ significantly in both size and latency. The memory structure schema, including memory buses and caches, is depicted in Figure 2.7.

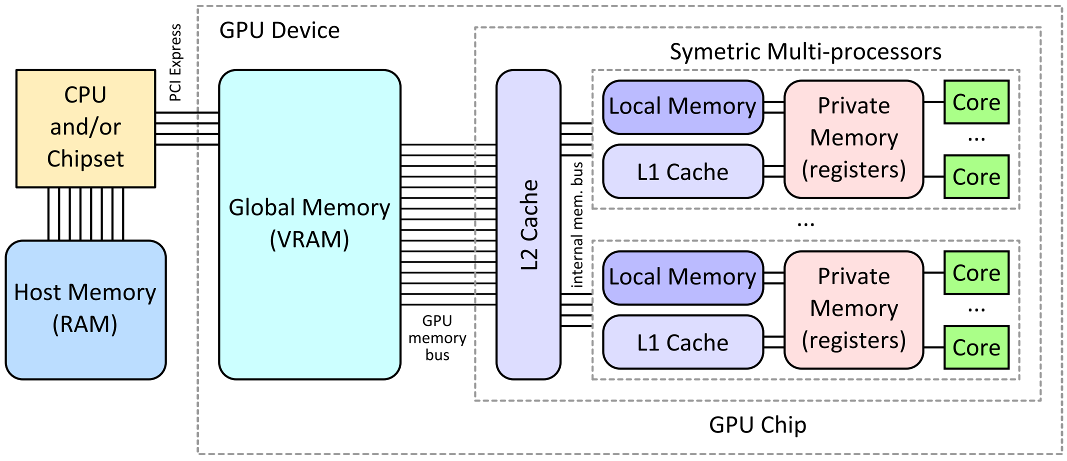


Figure 2.7: Structure of memory spaces on GPU

As we have already established in Section 2.2.1, the *global memory* is placed independently on the GPU card. It is several gigabytes large and connected to the GPU chip via an internal memory bus. This bus has a throughput several times higher than QPI, HT, or the integrated memory controller of the CPU, but we have to bear in mind that the GPU memory bus has to feed data to many more processing units. The throughput is achieved mainly by the width of the bus, which has usually at least 256 bits.

The memory controller also connects the global memory with the PCI-Express bus. When the host system wants to exchange data with a GPU device, the data are stored to or loaded from the global memory. Fortunately, current global memory controllers are capable of conducting data transfers to the host system and to the GPU core chip simultaneously, so the memory transactions and the computations can overlap.

The GPU chip is equipped with a transparent L2 cache (768 kB on Fermi) which is shared amongst the SMPs. It caches data from the global memory to reduce the number of data transactions on the external bus.

Each SMP has a small amount (64 kB) of integrated memory shared amongst the cores. This memory is divided between the *local memory*[4] and the transparent L1 cache. The exact division can be configured, but we have usually used 48 kB of local memory and 16 kB of cache.

The local memory is very important for code optimization. It is quite small, but on the other hand it is almost as fast as the registers and it is accessible by all the cores. If the code running on the SMP is optimized to use local memory as manually managed cache or for shared intermediate results, it has significant positive impact on the processing speed.

The *private memory* belongs exclusively to a single thread and corresponds to the registers of a GPU core. In fact, the registers belong to the SMP (32k of 32-bit words on Fermi) and they are allocated for the threads assigned to the SMP to create an illusion that each thread has a processing core for itself and avoid register saving/restoring when the warp is switched. If we use 512 threads in the group (which is also the current maximum), each thread will only have 64 words of private memory. The compiler may attempt to avoid the problem of limited private memory by spilling the private data to the L1 cache; however, this comes with a serious performance hit due to the increase in memory traffic and instruction count.

Finally, the *constant* memory is a small block of memory dedicated for immutable data shared amongst the threads. It has very limited size (64 kB), but since the data are constant, it can be cached very well, thus accessed quite fast.

2.2.4 Memory Performance Issues

There are two important issues concerning the architecture of the GPU memory that strongly affect the performance:

- the global memory data transfers
- the local memory organization

Data transfers between the global memory and the GPU chip are performed in transactions. Each transaction transfers an aligned data block of fixed size.

[4]NVIDIA designates this type of memory the *shared memory*. It is also used for texture data in 3D graphic, thus it was formerly denoted the texture memory or the texture cache. The term local memory comes from OpenCL specification, so we will hold to that to avoid ambiguity with the algorithm descriptions.

If the caching is turned on, each transaction transfers a cache line of 128 bytes. The 128 B corresponds to 32 4-byte words, so if all the cores in the warp read or write 4-byte words from within a 128 B aligned memory block, the data are transferred in one transaction. If the memory access pattern is less coalesced, it breaks down to multiple transactions, 32 in the worst case scenario.

The second issue is the organization of the local memory. The local memory is usually accessed by all 32 cores at once so it must be designed for highly parallel access. Since one memory controller would present a significant bottleneck, the memory is divided into 32 *banks*. Two consecutive 4-byte words are in two consecutive banks (modulo the number of banks). Banks operate independently, so if two threads access data in different banks, the operations are performed concurrently. On the other hand, if two threads access data in the same bank, their operations are serialized and the whole warp is delayed. In a special case when multiple threads read exactly the same word from the local memory, the memory controller broadcasts the value to the threads in one step. The banking principle is depicted in Figure 2.8.

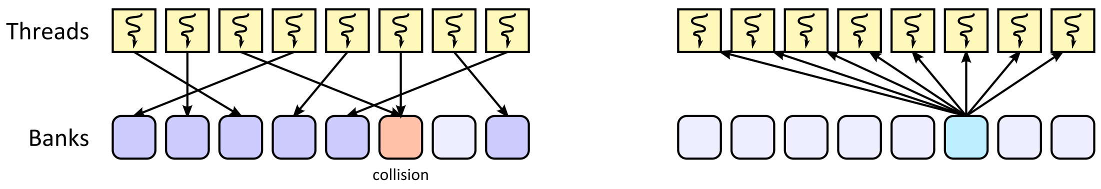


Figure 2.8: Threads accessing local memory randomly (on the left) and the broadcast optimization (on the right)

The issues described above lead to an optimization technique frequently employed in GPU programming. In most cases, the data should be organized as a *structure of arrays* instead of an *array of structures*. We can demonstrate the benefits on a simple study case.

Let us have a large number of 2D point pairs for which we want to compute the Euclidean distance:

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

On a CPU, we would organize the data as an array of structures, where each structure contains the pair of 2D coordinates (values x_1, y_1 and x_2, y_2). This practice is naturally embraced by the programmers as it is more comprehensible in the code and the CPU will deal with it quite well. If we use this organization on a GPU (assuming all values are 32 bit floats), we will create a few small problems. In the following example, the variables are always denoted x_1, y_1, x_2, and y_2 even though each thread is assigned its own set of these variables.

When a warp loads x_1 values form the global memory, the data are stridden since there is y_1, x_2, and y_2 value between each two adjacent x_1 values. This load will result into four memory transactions instead of one. Thanks to the L2 and L1 cache, the problem would not be so severe as the subsequent loads of y_1, x_2, and y_2 values will be performed from the cache.

If the input point coordinates are stored in the local memory (e.g., as an intermediate result produced by the previous computational step), the situation gets much worse. When a warp accesses each variable, the values are spread amongst 8 banks instead of 32. There are 4 threads competing for access to each of these 8 banks, thus the whole process is slowed by the factor of four.

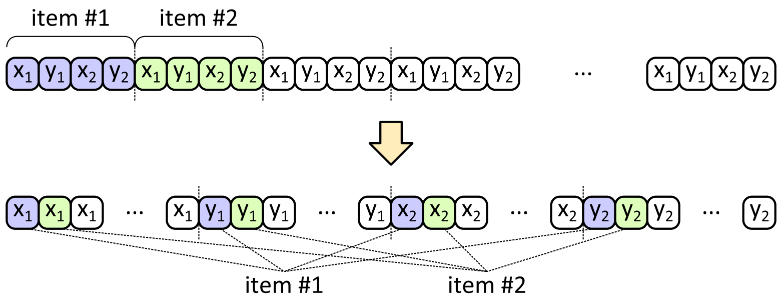

item #1
item #2
x_1 y_1 x_2 y_2 x_1 y_1 x_2 y_2 x_1 y_1 x_2 y_2 x_1 y_1 x_2 y_2 … x_1 y_1 x_2 y_2
x_1 x_1 x_1 … x_1 y_1 y_1 y_1 … y_1 x_2 x_2 x_2 … x_2 y_2 y_2 y_2 … y_2
item #1
item #2

Figure 2.9: Data organized as an array of structures and structure of arrays

We can organize our data as a structure of arrays (the difference is depicted in Figure 2.9). If we do so, the loading from the global memory will result in one memory transaction per each variable, thus producing more stable workload. In case of the local memory scenario, two consecutive x_1 variables are in consecutive banks, thus accessing them would not produce any bank conflicts.

2.2.5 GPU Programming

There are currently three frameworks that can be used for parallel GPGPU programming – CUDA [36], AMD Accelerated Parallel Processing SDK [37], and OpenCL [38]. The CUDA is a proprietary solution of the NVIDIA company and it is working on NVIDIA hardware only. On the other hand it is well established and quite easy to use, as it is designed solely for GPUs. The APP SDK (originally named *Stream SDK* or *Close-to-Metal*) is a proprietary solution for AMD devices. The AMD solution arrived later than CUDA and it did not reach the same popularity.

OpenCL, on the other hand, is a generic framework for parallel computing designed by the Khronos consortium representing many large companies. It has several implementations for GPUs (from NVIDIA, AMD, or Apple), so it can be used with various devices from various vendors. Furthermore, it encompasses also other parallel devices, such as multi-core CPUs, IBM Cell cards, etc. We have used the OpenCL in most of our experiments since we prefer an open solution to the private one. However, the algorithms and the optimization techniques presented in this work can be applied for any CUDA or APP implementations as well.

We briefly describe basic principles of the OpenCL framework as some of these principles are quite important for the algorithm design. The framework itself has two parts. On the host side, the *OpenCL runtime* provides an API with bindings to various languages that allow programmer to detect parallel devices and use them. For the devices themselves, the framework defines an *OpenCL language*, which is a subset of C99 language for programming kernels, and a list of built-in functions that must be implemented on the device (thread identification, atomic operations, mathematical functions, etc.).

OpenCL Runtime

The host-side libraries provide mechanisms to detect parallel devices and list important information about them, such as the number of computational cores, operating frequencies, or the amount of internal memory. When a parallel device is detected, the OpenCL can compile kernels directly for it. Runtime compilation produces much more efficient code, which is optimized for the target device. On the other hand, the kernel compilation itself takes some time. It does not usually bother us gravely as this time can be hidden in the start-up time of the parallel application or in the initialization procedures like the SQL `PREPARE` statement.

The OpenCL also manages the memory of the parallel device. The memory is allocated via memory objects. A memory object (buffer) is a logical entity that represents a continuous block in the memory of the device. The user cannot define, how or where the object is allocated as these details are implementation specific. The OpenCL defines a set of functions for transferring data to and from the buffers and the buffers can be assigned to kernel arguments in the form of global memory pointers.

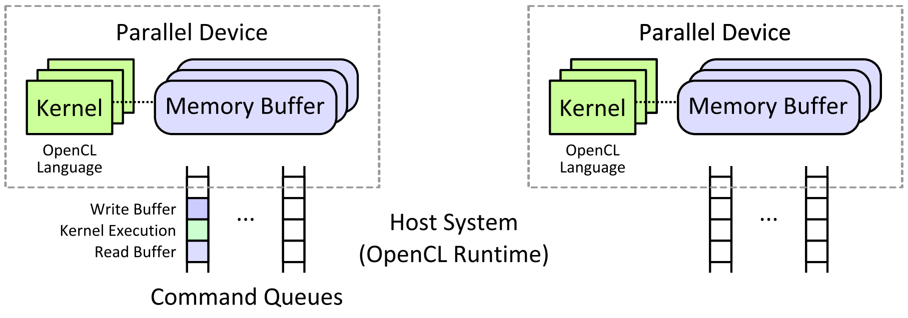


Figure 2.10: A model of OpenCL runtime entities

All operations performed on the device are issued through a *command queue*. One or more command queues can be created for each detected device. The most common commands are the read/write operations that manipulate data in memory buffers and the kernel executions. The whole schema is depicted in Figure 2.10.

Kernel Execution

When a kernel is executed, it is provided with a *global work size* and a *local work size*. The global work size defines the total number of work items spawned, while the local size specifies the number of items in a group. The work items directly correspond to the threads and the work group is in fact the thread group on the GPGPUs. All the work groups must have the same amount of threads, hence the global work size must be divisible by the local work size.

The work items can be organized into one, two, or three dimensional space and the global work size and the local work size values are specified for each dimension independently. The OpenCL language defines specific built-in functions that allow the work item (the thread) to get the values of work sizes, the work dimension, and the index of the item among the global work load and within the work group.

OpenCL Language

The OpenCL language is a subset of the C99 language [39] with some extensions. The most important differences are:

- Each pointer has an additional type (global, local, or private) that defines its address space (as we have already described in Section 2.2.3).
- The kernel cannot allocate memory. Private memory is assigned automatically to local variables, local and global memory must be preallocated before the kernel is executed.
- There is no stack. All function calls are inlined, recursion is not permitted.
- Fixed-size vector data types are introduced. Each basic type (like `int` or `float`) has a corresponding vector type of length 2, 3, 4, 8, and 16 (e.g., `uint4` is a vector of four unsigned integers). All basic arithmetic operations defined for the scalar types are defined for the vector types as well. These operations may be translated into fewer instructions if the architecture supports them.

Finally, we have to keep in mind that items in a group can be executed in SIMT or virtual-SIMT fashion. Therefore, an extensive usage of if-statements or while-loops ought to be discouraged.

3. Task Scheduling

In this chapter, we address several issues of the task scheduling. After a general overview of the topic, we focus on two specific problems of scheduling – the blocking tasks and the hybrid CPU-GPU tasks. We propose practical solutions for these problems and evaluate their impact experimentally.

3.1 The Problem of Scheduling

Task scheduling is one of the most important issues in any parallel system. If approached from a wrong direction, it has the ability to significantly hurt the overall performance. It may also have a direct impact on other system attributes, such as the utilization of processing units or the memory allocation, thus having serious influence on the power consumption of the whole system.

From the general point of view, there are many qualitative aspects of a scheduler, especially:

- *latency*
- *throughput*
- *fairness*
- *overhead*
- *scalability*
- *hardware utilization* (which implies power consumption)

The importance of these aspects differs with each system. For instance, the process (or thread) scheduler of an operating system is most concerned with latency and fairness [23, 22], while the tasks scheduler of a parallel framework for high performance computations [27, 40] requires high throughput and small overhead.

As this topic is very broad, we need to narrow our span. This work focuses on the computational parallel frameworks designed to process large datasets. Therefore, we do not consider fairness nor latency, as we expect that all hardware resources (CPUs, GPUs, . . .) are allocated solely for the application at the time. As we have restricted the domain of the problem, we can discuss some generic attributes of the task schedulers designed for our purposes.

A scheduler should focus mainly on *throughput* and *scalability*. The throughput defines the size of the data that can be processed at a unit of time or the time required to solve a problem of a fixed size. The scalability reflects the overhead and the limits of parallel processing. It helps us predict the future and determine, how would the same application work on newer hardware with more processing units. We also need to monitor hardware utilization closely since idle computational units usually suggest some room for improvement.

Furthermore, a scheduler should be *non-preemptive*. Preemptive scheduling assumes that we can suspend running tasks and resume suspended tasks, so we can execute more tasks than there are available processing units in a quasi-parallel manner. Such execution may be beneficial for interactive applications, where multiple processes need to interact with the user or with an external device. On the other hand, the suspend/resume operations are quite expensive and they do not help in case we need to wait for all the tasks to complete.

The problem of the uniform task scheduling has been investigated thoroughly. We provide an overview of the related work in the following section (3.2). Our contribution is divided into two parts. In Section 3.3, we address the problem of blocking tasks that may disrupt the occupancy of available computational units. Section 3.4 focuses on a special case of blocking tasks that raises in hybrid CPU-GPU systems. The experimental evaluation of both solutions is presented in Section 3.5.

3.2 Related Work

As the topic of task scheduling is quite broad, we divided our overview into three sections. First, we revise the task scheduling in general (Section 3.2.1) and establish basic principles of static and dynamic scheduling as well as the problematics of hybrid CPU-GPU scheduling. In Section 3.2.2, we describe the Intel Threading Building Blocks framework and its task scheduler. The TBB offers one of the best schedulers for multi-core CPU systems as it implements state-of-the-art methods and techniques. Finally, we present the Bobox system [40, 41] (Section 3.2.3), a highly parallel framework for data processing being developed at our departement. The Bobox aims at a more specific problem domain, thus it can employ more optimizations than TBB.

3.2.1 Task Scheduling

Let us define the terms first. A *task* is a part of work performed by the program. It comprises both code and data, i.e., the inputs and the algorithm that processes the inputs. Tasks usually have some dependency constraints, which ensure that a task cannot be scheduled before all its preceding tasks have finished. Tasks are very flexible as they can be used to exploit various types of parallelism. We can divide an algorithm to steps and let each task perform one step (thus achieving a pipeline), generate multiple tasks with the same code that work on different parts of the inputs (data parallelism), or even create a complex combination of those types.

The term *scheduling* describes an algorithm that allocates computational units and assigns tasks to them. Task scheduling is similar to the process scheduling employed by the operating system [23, 22] to allocate computational resources for running processes. Task scheduling is prefered to the process scheduling in data processing frameworks as the tasks have much lower overhead than processes or threads. It is also much faster to assign task to a worker thread than to switch threads on a CPU core.

Various Types of Scheduling

Even though we have restrained the boundaries of the scheduling problem, there are still many variations [42, 43, 44] that depend on the additional assumption we make about the tasks or the hardware and whether we have some external constraints like hardware limitations or deadlines. We examine several such assumptions and constraints to demonstrate various approaches to scheduling.

First, we can discuss the properties of the tasks being scheduled, especially their creation and length estimations. If all tasks are known in advance and their length can be estimated or even calculated with reasonable accuracy, a schedule can be computed in advance. Finding an optimal schedule is basically a variation of the *knapsack problem* [45, 46], which is a *NP-hard* problem [47]. Various approximation algorithms exist, usually based on the greedy approach [48, 49]. Computing the schedule is rather expensive, so this approach is only optimal for small numbers of long-lasting tasks.

The knapsack optimization (or its approximation) works fine until task dependencies are added. With the dependencies, the tasks form a directed acyclic graph (DAG) and they need to be scheduled with respect to the dependency satisfaction constraints. This type of scheduling is usually solved by the *Critical Path Method* [50]. The CPM identifies the longest path in the task graph and marks tasks on this path as critical. Knowing the critical tasks helps the scheduling (as we can, for instance, assign them to a better hardware), and can also be used in some optimizations planing. This method is rarely used for task scheduling in parallel systems, but it has its applications in project management.

Additional problems arise when deadlines are introduced to the system [51, 52]. We recognize *soft deadline* and *hard deadline* real-time systems. A soft deadline denotes a deadline that can be missed, but such event should be as rare as possible. Real-time processing of digital signals is a good example [53]. Hard deadline systems [54] are prohibited to miss any deadlines once they accept a task. This approach is typical for mission critical systems, such as onboard control systems of aircrafts, boats, or vehicles.

Even though there are many shades of grey in the realms of scheduling, our choices are rather limited since the parallel and distributed systems for data processing have quite specific properties. First of all, we usually cannot make any assumptions about the tasks nor the hardware. The tasks are too short and too diverse, so their computational time is hard to estimate with reasonable margin of error. On the same note, the tasks are not all enumerated in advance since their number may depend on the size of some intermediate result. Even the number of the computational cores allocated for the framework may not be constant as we show in Section 3.3.

From now on, we consider only systems with following properties.

- There are no assumptions about the length of the tasks and the tasks are emerging on the fly.
- The tasks are processed by the worker threads while the dispatching cost is very small.
- The tasks may have dependencies that enforce their processing order, but no other constraints (e.g., deadlines) are applied.

The Dynamic Approach

Previous methods could be called the *static approach* as they create the schedule once and the tasks are executed according to that schedule [55]. When new tasks emerge dynamically, this approach is becoming less efficient as we need to reschedule the tasks from time to time and the scheduling itself takes significant amount of time.

Another approach is to employ some kind of *dynamic scheduling* [56, 57, 58], which deals with tasks and systems where properties are unpredictable or even changing in time. The dynamic scheduling is expected to schedule tasks on the best effort basis with as little overhead as possible.

There are several naïve ways how to implement dynamic approach. We can hold a pool of tasks from where the tasks are dispatched to (or taken by) the worker threads. Unfortunately, as the number of workers increase, the task pool inevitably becomes a systemwide bottleneck. An opposite approach would be to assign tasks to workers immediately, in a round robin fashion for instance. Such algorithm would produce seriously imbalanced workloads and it would not deal with some problems like the varying number of workers.

In order to avoid bottlenecks, imbalance of the workload, and excessive overhead, the task pool must be split into multiple parts – usually as many as there are workers, so each worker has its own pool. A load balancing protocol must be employed to ensure that all workers have work to do while there are still tasks to perform. One of the easiest methods, which has minimal overhead and very good performance, is the *task stealing* technique. This technique is employed by Threading Building Blocks [27, 28] and we describe it later in Section 3.2.2.

A more sophisticated approach is to perform *task balancing* every now and then. In each balancing step, all tasks from all pools are taken and redistributed evenly in the pools. We can perform such action periodically, under specific conditions (like when serious imbalance is observed), or when one of the workers runs out of tasks. Even though this method seems to be more prudent in planning, it does not perform much better than task stealing and it has a significantly greater overhead.

Hybrid CPU-GPU Scheduling

Scheduling principles described so far were designed mostly for symmetric multiprocessing systems or at least homogeneous systems. Emerging parallel architectures like GPGPU or Intel MIC are introducing heterogeneous platforms for parallel computing. Such platform is usually equipped with multi-core CPU and additional parallel devices connected to the host system. Scheduling in these systems brings new problems, such as additional communication or synchronization, and a more complicated load balancing.

One of the first attempts to explore the problems of hybrid CPU-GPU task scheduling was made by Wang et al. [59]. They have presented a collaborative computing model to bridge the gap between the CPU and GPU utilization. Jablin et al. presented a framework for an automatic management and optimization of CPU-GPU communication [60]. This framework is designed to simplify the design of GPGPU applications while optimizing data transfers between the host system and the parallel device.

A slightly broader view to the problem was presented by Jimenez et al. [61] as they addressed the issue of sharing parallel devices among multiple users. Their solution proposed a predictive scheduler that estimates demands of the users based on the performance history. The work of Shirahata et al. [62] expanded the problem of scheduling even further – to the realms of computational clusters. Their solution is based on the MapReduce approach [63], but it utilizes GPU devices in addition to multi-core CPUs.

Related work revised above focuses on specific problems of the hybrid scheduling with slightly different objectives than we have. The closest solution we found was the StarPU [64] framework, which is an unified runtime task scheduling system for heterogeneous architectures. It evaluates a task graph and schedules tasks on both CPUs and GPUs while it attempts to optimize data transfers and communication. Unfortunately, the project aims strongly at user friendliness. We needed a lower level of control in the task scheduling as our most important objective is peak performance and we are willing to sacrifice some comfort in order to achieve it.

3.2.2 Intel Threading Building Blocks Scheduler

The Intel Threading Building blocks library [27, 28] has already been introduced in Section 2.1.3. In this section, we take a closer look on the TBB scheduler and explain how it works in detail.

Task Graph

The scheduler executes a *task graph.* A task graph is a tree where tasks represent vertices. Each task has one oriented edge pointing to its *successor*, except for the *root* tasks that do not have any successors. Furthermore, each task has a *refcount* variable that counts the number of other tasks which have this task as a successor. When a task finishes, it automatically decrements the refcount of its successor and when the refcount reaches zero, the task is inserted into the *ready pool* to be scheduled. As will be shown later, the refcounts need not correspond directly to the number of incoming edges in the task graph.

We can define a *depth* of a task as follows. The root tasks have depth equal to zero. Every other task has the depth of their successor plus one. In the first implementation of TBB (version 2.x), the depth was also stored in the task structure along with the refcount and the reference to the successor. The current TBB version has the scheduler streamlined so the depth is no longer necessary, but we still define it, so we can describe and compare both versions later.

The task graph changes dynamically as new tasks are being created. The whole idea stands on a simple fact that tasks can spawn other tasks, thus the creation itself can run in parallel. The dynamic task creation could also have a significantly smaller memory usage. Furthermore, the system can process even problems that cannot be easily measured and divided into tasks in advance.

Figure 3.1 captures an example of a task graph. Tasks A, B, and D are already allocated, but they are waiting for their children to finish. Tasks C, E, F, and G are ready to run or running, depending on how many threads are available. The grey tasks $(H, \ldots, N)$ have not yet been spawned and they only illustrate, how will the computation continue.

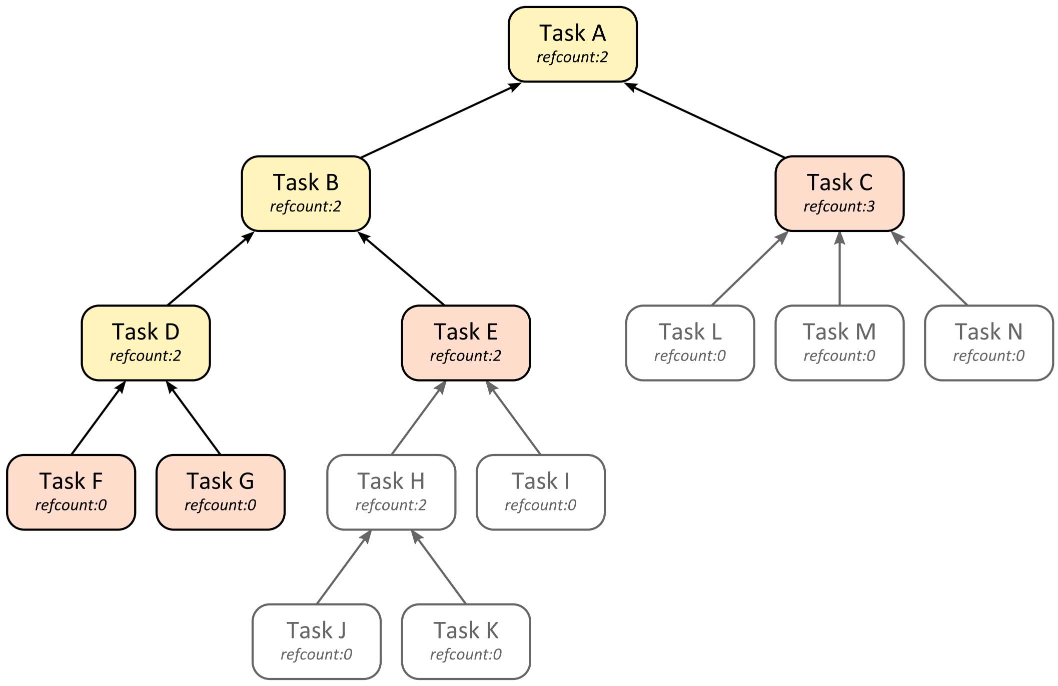


Figure 3.1: An example of a task graph

When a task spawns its children, there are two ways how to process them and how the graph will unfold:

- depth-first
- or breadth-first.

The *depth-first* traversal is ideal for a serial execution. It has the lowest memory consumption (assuming the graph is finite), because only one branch is unfolded at any given time. It also makes the best use of CPU caches since the most recently created task is also likely to be the hottest in the cache. The depth-first approach is depicted in Figure 3.1, where the leftmost branch has been fully unfolded. On the other hand, the *breadth-first* approach unfolds the graph in the quickest possible way, thus providing the greatest potential for parallelization.

Thread Pool

The tasks are processed by threads in the thread pool. By default, the TBB framework creates as many threads as there are logical CPU cores (i.e., it exploits hyper-threading as well). More precisely, it creates one less worker threads than there are CPU cores so that the main thread still has its own core. This way, the main thread can do other work while the workers are processing tasks and when the main thread tries to wait for the workers, it may become a worker too, so it helps the workers to finish the tasks.

Each thread has its own *double ended queue* (deque) for the tasks. We denote the ends of the queue the *top* and the *bottom*. When a new task is spawned by a thread, it is pushed to the bottom of the deque. Hence, the oldest task is at the top and the youngest is at the bottom. When a thread finishes a task, it needs to find another task to execute. It does so by application of following rules:

1. The finishing task can yield a reference to another task which should be executed next. This is actually called a *scheduler bypass* as it allows the programmer to interfere with the regular scheduling mechanisms.

2. If the task deque is not empty, a task is popped from the bottom and executed. As we mentioned before, new tasks are pushed to the bottom, so this case corresponds to the depth-first traversal of the graph.

3. If the deque is empty, the thread attempts to steal a task from the top of the deque of a random thread. That corresponds to the breadth-first unfolding of the graph. If the stealing fails, the thread attempts to steal again until it succeeds[1].

We can summarize this strategy as depth-first work and breadth-first theft. It makes each thread proceed to the depth, thus better utilizing the caches and reducing memory demands, but it simultaneously allows occasional breadth expansions to occupy all the workers.

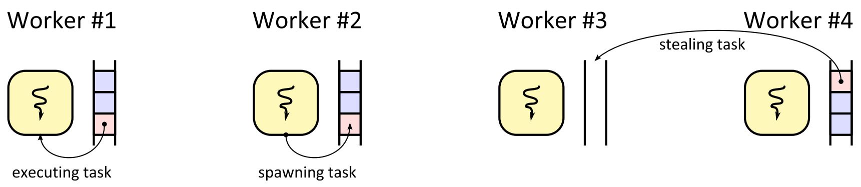


Figure 3.2: A thread pool of worker threads

The older version of the scheduler used a more elaborated pool of ready tasks. The pool comprised an array of lists. The array was subscripted by the depth of the task and the lists were treated as stacks. When a task was spawned, it was inserted into the front of the list in the corresponding depth. The rules for getting tasks out of the pool were very similar to the rules described above. The thread took the first task in the deepest occupied list of its pool, or it attempted to steal the first task of the shallowest list of a random thread.

We can observe, that if the depth of a task is determined as the depth of its parent plus one, the tasks in the deque are in fact ordered by their depths (shallowest at the top and deepest at the bottom). Hence, the current scheduler, which employs simple deques, works as fine as the older scheduler, but it managed to save some memory by omitting the depth values from the tasks.

Task Programming Patterns

Before we explore the most common programming patterns for the task scheduler, we need to clarify, how the tasks are planned (i.e., how they get into the deque of the ready tasks). There are three ways, how the task may enter the ready pool:

- A task is spawned *explicitly* when its parent task (or the main thread) invokes the `spawn` method of the Threading Building Blocks API.

[1]Our description was taken from the TBB documentation [28]. However, we have observed that additional optimizations were implemented since the worker threads are suspended in case there are not enough tasks in the whole system.

- When a task has been marked for reexecution, it is re-enqueued on termination.
- The refcount of a task is decremented every time one of the tasks referring to it terminates. A task is enqueued for execution when this counter reaches zero.

The easiest pattern for spawning children is the *blocking style*, which is depicted in Figure 3.3. It corresponds to the standard recursive implementation of the divide and conquer paradigm. The parent task spawns its children and waits for them to terminate by calling the `spawn_and_wait` method. This method also ensures, that the worker thread being used by the parent task is temporarily vacated, so it can process the child tasks. The state of the parent task remains intact and it can easily resume when the method returns.

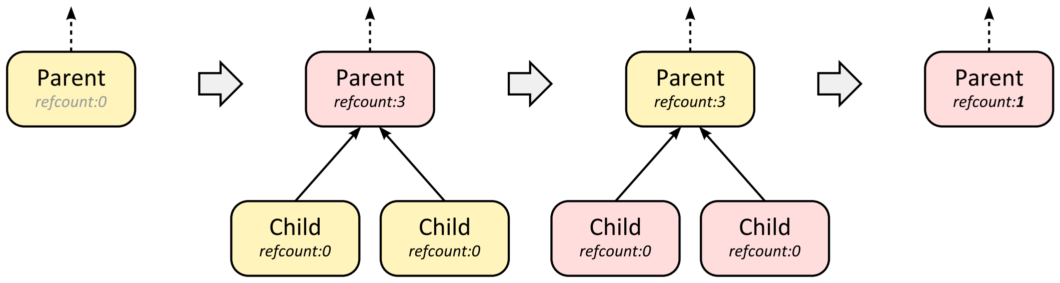


Figure 3.3: Spawning child tasks in the blocking style

Before the parent task spawns its children, it must update its refcount correctly. We need to ensure that the task is not reentered automatically to the ready pool, as it would be executed again from the beginning instead of being resumed form the waiting method. The refcount has to be set to $k+1$ in case k children are spawned. The "+1" guard ensures that the task will not get reexecuted. After all k child tasks terminate and the refcount is decremented k times, it remains 1. Thus, it never reaches zero and the task is not reentered to the ready pool. Instead, the termination of all the children causes the spawning method to return, and the parent task resumes its work.

Though the blocking style is quite convenient for the programmer, it has some opportunities for improvement. The most tedious is that the parent task keeps its state on the call stack. Furthermore, the parent task cannot be stolen from its thread since it is considered to be running. We can use the *continuation style* to deal with these problems.

The continuation style is depicted in Figure 3.4. The parent task creates child tasks and also a *continuation task*. The continuation task is the successor of all child tasks and its refcount is set to the number of children. Hence, it is automatically enqueued to the ready pool when all the children terminate. The parent task transfers its state to the continuation task, so the continuation can pick up where the parent task finished. After spawning all the children, the parent task terminates, thus it vacates the call stack.

The continuation style works better with the scheduler as it splits the original work of the parent task into two tasks. The continuation task can be treated separately, even get stolen by another thread. On the other hand, it is usually

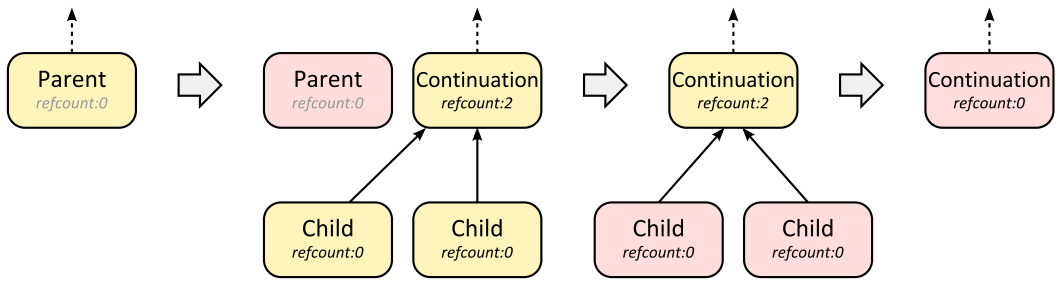


Figure 3.4: Spawning child tasks and their continuation task

more difficult to use as the programmer is responsible for transferring the state of the parent to the continuation task. To optimize further, we can recycle the parent task as the continuation task in some cases. If we do so, we save some memory allocation and data copying.

3.2.3 The Bobox System

Bobox [40, 41] is a highly parallel framework designed specifically for the needs of the database management systems or similar systems that query or process large amounts of data. It works on common CPU symmetric multiprocessors, thus it can be used with current personal computers or NUMA servers.

One of the main objectives is to make the system easy to use. Pieces of code that are written by the programmer are always executed in serial and they are designed not to interfere with each other. Therefore, the programmer needs to focus on the semantics of the problem, not the parallelization details like the scheduling, the synchronization, or the deadlock avoidance. We also believe that most of the user-defined functions (like sorting or joining) can be written in a generic way, so there is a high potential for code reusability.

So far, the Bobox framework was used for several applications. One of the most important is a SPARQL evaluation system [65, 66] that processes RDF data [67]. It has also been used for querying semi-structured data (XML) by the means of the XQuery language [68] and its TriQuery extension [69].

Bobox Fundamentals

The Bobox approach to parallelism is based on the ideas of nonlinear pipelines and concurrent data streams processing. The framework processes one or more *models*. A model (depicted in Figure 3.5) is a directed graph with *boxes* as vertices and edges that specify the connections between outputs and inputs of the boxes. The boxes are independent operators that process the data. The code of each box is executed serially, which is convenient for the programmer, but the boxes are planed by multiple threads, thus they can run in parallel. Data are transmitted along the edges as packages called the *envelopes* and the system ensures that these transfers are thread safe.

A box is an object with an internal state and one *processing method*. It can also have an arbitrary number of inputs and outputs, while the inputs and outputs are perfectly paired in the model. When the processing method of the box is executed, it usually consumes the input envelopes and creates some output

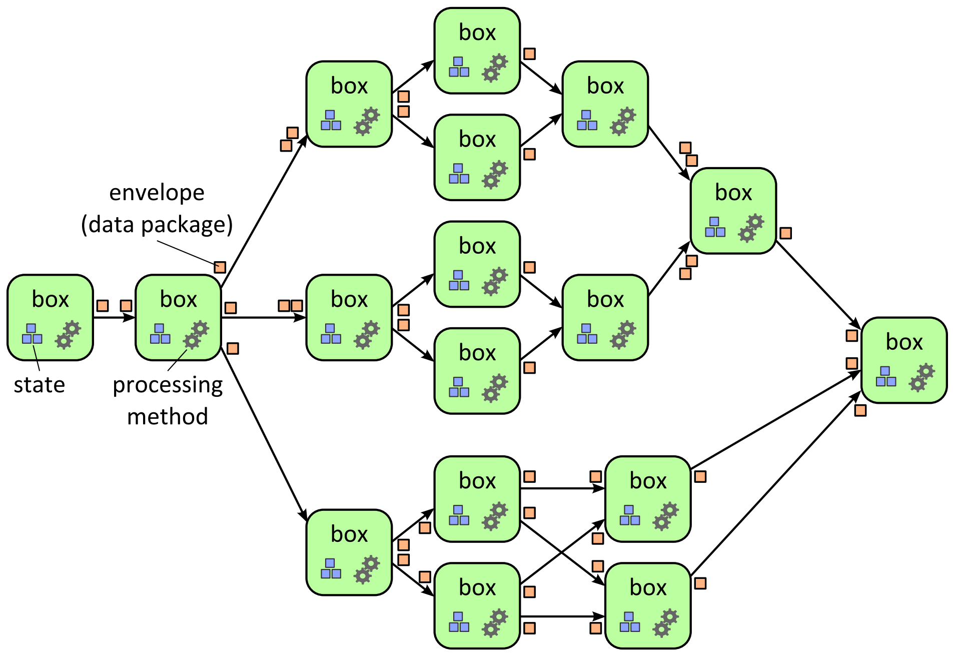


Figure 3.5: Bobox model example

envelopes. It can also modify the internal state of the box. Inputs and outputs are buffered and the framework ensures the envelopes are transferred from the output buffers to the connected input buffers automatically and in a thread-safe way.

The graph contains at least one box that generates data. These *initial* boxes usually do not have an input, but they generate or load the data (e.g., from a persistent storage). Analogically, there is at least one box that usually does not have any outputs and it consumes the data. These *sink* boxes are programmed to save the results back to the persistent storage or yield them to a Bobox user.

The envelope is basically a data table (an array of records), but the data are organized in a column-oriented manner (i.e., as a structure of arrays). Each input and output has a descriptor that defines the number and types of columns of the envelope they can receive or send respectively. A connection can be made only between an input and an output with the same envelope descriptors.

The column-oriented organization of the envelopes was originally designed so the system may benefit from the SIMD instructions of the CPU. Another reason was that we can copy the entire columns between envelopes just by passing on pointers in case some of them do not change. Finally, this data representation is suitable for other highly parallel systems, such as GPGPUs.

Top-level Architecture

The architecture of Bobox is depicted in Figure 3.6. It demonstrates the usage of the Bobox as a part of a database management system, which is what the Bobox is designed to do in the first place. The system receives *queries*, processes them, and yields their results. Multiple queries can be processed simultaneously.

The query is parsed by a *frontend*, which is basically a compiler. The frontend compiles the query and issues a new *request* to the system. A request is just a logical concept that helps us differentiate tasks that process separate queries

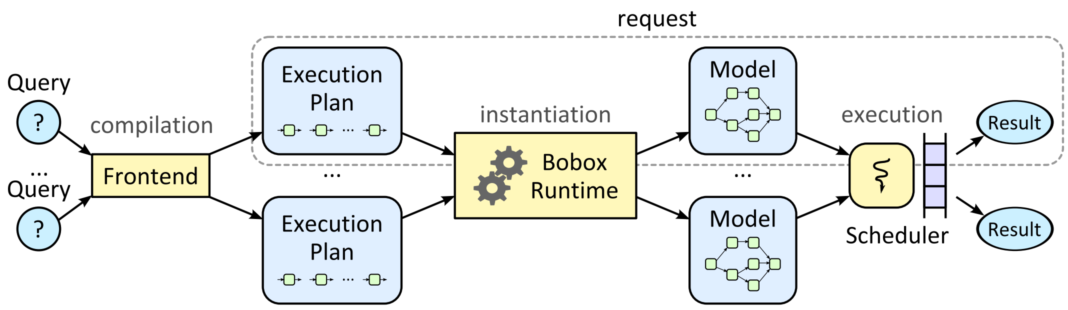


Figure 3.6: The Bobox architecture

and correctly match computed results to the original query when the work is completed.

The request is represented by an *execution plan*. The execution plan is a formal description of a model (depicted in Figure 3.5) and it is handed to the *Bobox runtime*. The runtime instantiates the execution plan and creates a corresponding *model*. The model is yielded to the task scheduler along with an initial task and the scheduler starts processing the model.

Bobox Scheduler

The most essential and the most interesting part of the Bobox framework is its scheduler. It is inspired by the TBB scheduler, but it is adapted for more specific tasks produced by the models and it is better optimized for cache usage and NUMA systems [70].

The Bobox scheduler (depicted in Figure 3.7) uses a thread pool which has the same number of threads as there are logical CPU cores available. Each thread is aware of its location within the CPU hierarchy – i.e., on which logical core, physical core, and NUMA node it runs.

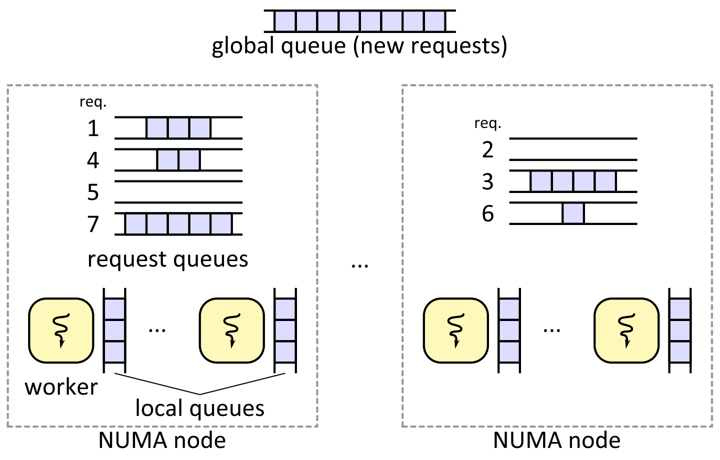


Figure 3.7: The Bobox scheduler design

Many principles of the scheduler are similar to the TBB scheduler, so we focus mainly on the differences. The tasks for the scheduler are in fact planned executions of the box processing methods. A box is never scheduled by multiple threads since the code in the box method is not expected to be reentrant.

Each task is tagged by the *request id*, which also defines the model where the box belongs to. Request ids are generated sequentially, so we can easily recognize younger and older requests. The basic strategy of the scheduler is to keep all the tasks of one request on one NUMA node, but not to reduce parallelism if the number of requests is low. Each request is assigned to one node at the beginning. A load balancer monitors utilization of all nodes and reassigns the requests or makes them shared when necessary.

There are three types of task queues in the system. Initial tasks of new requests are placed in one *global queue*. Each NUMA node has a list of requests the node is working on and each of them has a *request queue* of tasks. Finally, each thread has a *local double-ended queue* like the TBB worker threads.

The tasks for the scheduler are not spawned directly like in TBB. Instead, each box may notify the scheduler that it wants to be scheduled on some important event, usually when an input envelope arrives or some space is vacated in the output buffer. The scheduler plans the task automatically, when such event occurs. An execution of the processing method of the box usually spawns some other tasks. The box may be rescheduled if it has some unprocessed input and adjacent boxes connected to the output may be scheduled if some envelopes are generated.

In most cases, it is better when the following boxes are scheduled immediately and on the same core, as their input envelopes were just generated and there is a good chance they are still hot in the cache. On the other hand the box which just finished need not run again immediately, as the state of the box is often quite small in comparison to the size of the envelopes. To deal with these priorities, we distinguish *immediate* tasks and *regular* tasks. The immediate tasks are pushed to the bottom of the local task deque of the thread, and they are treated like in the TBB. The regular tasks are pushed to the request task queue of the NUMA node.

The task stealing technique is employed in a similar manner as in the TBB with only minor modifications. Since the threads are aware of their location and the location of other threads, they try to steal from their closes threads first. That means an idle thread tries to rob its hyper-threading buddy first, then it tries to steal from threads that share some caches with its processor core, and finally it tries the remaining cores. The basic task stealing is restricted to one NUMA node.

When the basic task stealing fails, the thread takes the first task of the request task queue of the oldest task being processed by the current NUMA node. If there are no remaining tasks in the request queues, an initial task of a new request is taken from the global queue and this request is assigned to the NUMA node. In case there are no new request to be processed, the thread is suspended until the load balancer reassigns some requests to the node or until some new work emerges.

3.3 Blocking Tasks

Most of the parallel frameworks are designed to process computational tasks. That means each task is expected to utilize its processing unit to the best of the hardware abilities. On the other hand, complex systems sometimes require *block-*

ing operations[2] to be performed, such as I/O transactions to a persistent storage device or a communication with peer nodes in a computational cluster. If such operations are performed in regular tasks, it may lead to a serious underutilization of computational units.

We stated that the tasks are scheduled to worker threads non-preemptively. That means, once the task is taken by the worker, it cannot be interrupted or forcibly replaced by another task. When a blocking operation is executed within a task, the assigned worker thread is suspended by the operating system until the operation is resolved. If the worker pool contains exactly the same amount of threads as there are available cores (which is quite common case), one of the cores becomes idle even though it is capable of processing another task.

There are several solutions to this situation. We begin with revising some naïve solutions and follow by presenting our solution implemented in the Bobox system.

3.3.1 Naïve Solutions

First idea would be to increase the number of threads in the worker pool, so the system has at least as many running threads as CPUs even if some of the treads get suspended by the blocking operations. This solution would not require any modifications to the scheduler nor the tasks. On the other hand, this thread oversubscription creates additional work for the scheduler of the operating system, as it has to switch these threads on available cores evenly. The context switching of the threads has some overhead and it causes a measurable drop in the performance.

We can choose to create the substitute threads on demand to avoid their oversubscription. When a task needs to perform a blocking operation, it must create a new thread and insert it into the worker pool as a replacement. Then it enters the blocking operation and the newly created thread takes over the task processing. Unfortunately, a problematic situation arises when the blocking operation terminates and the task wants to resume the computational operations. It must somehow synchronize the entire pool, wait for one of the threads to terminate, and remove it from the pool. Furthermore, the creation and destruction of a thread is a quite costly operation and it should be avoided when possible.

Another possible solution would be to divide tasks into two groups – the *computational* tasks and the *blocking* task. Each group has its own thread pool to process the tasks. The computational thread pool has exactly one worker per available core. The blocking thread pool can have an arbitrary number of threads. We can choose its size based on how many blocking operation we would like to run in parallel, or we can design the pool to grow automatically when needed.

This approach works quite fine, but it is not very convenient for the programmer. The programmer must design the tasks to fall exactly to the computational group or the blocking group. Any heterogeneous task that would contain a sequence of blocking and computational operations must be divided into multiple tasks and the corresponding dependencies must be created so that these tasks are indeed executed sequentially.

[2]The blocking operations and blocking tasks in this section do not have anything to do with blocking style task programming described in TBB task programming patterns.

3.3.2 The Bobox Solution

We have described some of the direct approaches to the problem and their weaknesses. Our proposed solution tries to take the best of these methods while avoiding the problematic parts. Let us summarize what we have learned so far.

- In an ideal case, the number of threads executing regular tasks should be equal to the number of CPU cores. Less threads cause hardware underutilization, while more threads increase the overhead of context switching employed by the process scheduler of the operating system.
- The operating system scheduler and the principle of blocking operations cannot be easily modified or bypassed, so a replacement worker thread must be dispatched when a task performs a blocking operation.
- Repetitive creation/destruction of threads needs to be avoided as it is a costly operation. Thread pool should be used for both regular and replacement workers.
- The programming model needs to be sufficiently convenient for the user, otherwise it will not be used.
- The number of blocking operations running simultaneously (especially the disk I/O transactions) needs to be limited in some cases. Such limit should not collide with the ability to process regular tasks.

Attach/Detach Principle

Bobox uses two thread pools. A pool of worker threads that process the tasks and a pool of suspended backup threads which can be quickly resumed and added to the worker pool. The size of the worker pool is determined by the number of available CPU cores. The number of threads in the backup pool specifies, how many blocking operations can be pending simultaneously.

Two new functions are introduced in the API: `attach` and `detach`. These functions allow the programmer of a task to manipulate the number of worker threads in the system. The `detach` function should be called immediately before the blocking operation is invoked and the `attach` function needs to be called right after the blocking operation terminates. If used properly, there should be one active worker thread per available CPU core at almost all times.

When `detach` is called, the system resumes one of the backup threads and inserts it into the worker pool. The blocking operation can be invoked after that, so the current thread is in fact suspended and the resumed thread takes over in the computational work.

If there are no more backup threads, we need to delay the blocking operation but we also do not wish to reduce the number of working threads since there might be plenty computational tasks waiting to be processed. Therefore, the blocking task is suspended in the `detach` call and removed from the worker thread[3].

[3]We use *fibers* [71] on top of regular threads in order to manipulate their workload in user space.

The suspended task is placed into a queue designated for detached tasks where it waits until one of the backup workers becomes available. Though we have claimed that the tasks are always non-preemptive, an exception is made in the case of the detach call.

When `attach` is called, it notifies the scheduler that the size of the worker pool needs to be reduced by one and it immediately suspends the thread. The thread remains suspended until one of the running tasks finishes. The thread that was processing that task is removed from the worker pool, suspended, and inserted into the backup pool. After that the thread suspended in the `attach` call is resumed and it can continue with its task.

Additionally, when a worker thread is being suspended and moved from the worker pool to the backup pool, the detached tasks queue is checked. If the queue is not empty, the thread is not removed, but it takes a detached task from the queue and resumes it. The task is expected to execute a blocking operation immediately after it is resumed, so the thread gets suspended again.

Possible Improvements

There are a few possible improvements to this solution, but we have not implemented nor tested them yet. First is an automatic attach/detach invocation. There are two possibilities, how to achieve this. We can design an API for all blocking operations and encourage the programmer to direct all blocking calls via this API. Every blocking call through this API automatically invokes `detach` at the beginning and `attach` at the end. The second possibility is to perform a statical analysis of the code at the compilation time in order to identify the blocking operations and wrap them correctly with detach-attach calls. Such solution would be even more convenient for the programmer; however, it is based on the assumption we can identify all blocking tasks by the code analysis.

The experimental results suggest that, different types of blocking operations may lead to different behaviour. More elaborate analysis of various types of such operations is in order. It might be beneficial to treat different types differently.

3.4 Hybrid CPU-GPU Scheduling

If the scheduling problem is expanded to the hybrid CPU-GPU systems, several additional problems rise. To achieve the best performance, both CPU and GPU has to be utilized optimally. The situation gets rather complicated, since the GPU tasks (i.e., kernel execution and memory transfers) have to be issued from the host system, hence they require some CPU time as well.

Theoretically, this problem should be taken care of by the GPGPU programming libraries like OpenCL. Unfortunately, the OpenCL implementation provided by NVIDIA, which we had at our disposal, is quite conservative and does not exploit all the possibilities the standard offers. We have observed that it underutilizes hardware resources even in situations, when there is no apparent reason for any restrictions. For instance, when independent kernel execution and buffer transfer operations are issued into one command queue, they are performed serially even in case the queue is marked *out of order*, thus capable of executing operations in parallel.

Finally, we need to point out that the data are packed into the GPU buffers and the GPU device has much more limited memory space than the host system. The data transfers between the CPU and the GPU usually require some additional data operation like gathering relevant data from host memory into compacted form for the GPU or scattering the results from the compacted form back to host memory. Sometimes, it is also necessary to reorganize the data, e.g., from an array of structures to a structure of arrays as demonstrated in Section 2.2.4. These transfers require significant amount of CPU time and they can delay the GPU tasks if not handled properly.

We have designed a task dispatching framework that deals with these kinds of problems and helps us develop GPGPU applications much more conveniently. The framework is built on top of the OpenCL and TBB primitives and it can be easily combined with the TBB library or the Bobox framework. It has been used in every our prototype implementation and it performed flawlessly.

3.4.1 OpenCL Framework

As our framework is built on top of the OpenCL framework [38], a brief revision of the key principles of OpenCL is in order. The API that OpenCL provides is quite simple and easy to use. The host program can detect parallel devices that support OpenCL, enumerate them, and retrieve basic information about them. There are three most important things we can do with a parallel device:

- allocate and managed memory buffers,
- compile kernels,
- and execute kernels in parallel.

Command Queues

Parallel device operations are controlled via *command queues*. A command queue is an OpenCL object, which is attached to exactly one parallel device, but one device can have arbitrary number of queues attached to it. The API provides queue functions to issue memory buffer read/write transactions and to commence kernel executions. All these operations can be either blocking (the calling thread is suspended until the operation concludes), or nonblocking. Nonblocking operations are working asynchronously and the API provides barriers, events, and waiting operations for synchronization.

Each command queue can be either *in-order* (which is the default), or *out-of-order*. The in-order queues perform their operations in the exact same order, as they were issued by the main thread. The out-of-order queues can reorganize the operations (except for the synchronization markers and barriers), or even execute them in parallel when possible. Operations in two concurrent queues attached to the same device do not have a well defined order and they can be executed in parallel as well.

The NVIDIA GPU devices with compute capability 2.0 and higher are equipped with two memory controllers, so they should be able to overlap host-device transactions with kernel execution. This overlapping is often quite important for the overall performance. Unfortunately, we were unable to achieve this effect using the out-of-order queues, even though the OpenCL specification allows it. Therefore, we have used only the in-order queues and attempted to achieve parallelism by other means.

Memory Management

Another important issue is the management of memory of the parallel devices. The OpenCL provides a mechanism for allocating buffers, which are accessible both by the host system and by the parallel device. A buffer is a continuous block of memory that is placed in the device if possible. It can be marked with several flags that configure basic properties, like whether the buffer can be read or written by a kernel. Even though these flags allows us to control some attributes, most buffer properties remain implementation defined. For instance, we cannot ensure that all allocated buffers are really present at the device, since the framework may swap them to the host memory when necessary[4].

There are two basic ways how to manipulate data in buffers from the host side. We can issue a read or write operation to the command queue. The command queue has to be attached to the same device where the buffer resides and another buffer must be provided in the host memory. Then the data are transferred from the host buffer to the device buffer in case of a write or from device buffer to host buffer in case of a read. The second option is to map the buffer to the host memory space. In that case, the host application will get a pointer to the beginning of the buffer and all standard memory operations are transparently synchronized.

Kernels

Finally, we need to address the kernel issues. The kernel is usually provided in source code form, so it needs to be compiled first. We can list the parallel devices for which the compilation is performed and the OpenCL framework automatically shield us from the fact that there may be multiple binaries of the same kernel in the system. The appropriate kernel binary is selected when the kernel is being executed.

Before a kernel is executed, its arguments must be assigned. Pointers to the global memory can be assigned to the memory buffers. Pointers to the local memory can be provided with a size value. A memory block of this size is allocated in the local memory and set to the pointer argument when the kernel is executed. Constant values are assigned to the remaining (private) arguments.

Execution of the kernel is issued via the command queue. The kernel identifier is provided along with the number of threads (and the dimensions for the thread IDs) and how many threads are in one group. We have described the thread spawning and the execution process thoroughly in Section 2.2.

3.4.2 Model Case Study

In order to understand the design of our framework, we have performed a model case study to analyse the problems and to gather the requirements for the frame-

[4]Actually, the newest OpenCL standard (1.2) added some support for the buffer migration, but it still remains mostly under the control of the framework.

work. We do not cover every conceivable scenario, but rather describe some of the problems encountered during the design of various types of GPGPU applications [12, 13, 72]. Each of the following models describes a possible use case of the framework.

Model Cases

- The most *simple case* is using a single GPU task. This scenario is possible only if all input, intermediate, and output data fit in the GPU memory. Multiple simple tasks may overlap or even run concurrently on separate GPU devices, but they solve different problems, thus they are completely independent from the CPU-GPU cooperation point of view.

- If the data of the GPU task do not fit the GPU memory, or they are being streamed, multiple GPU tasks are required. In this *iterative case*, the data are divided into blocks and all the blocks are processed by the same algorithm (GPU kernel).

- The GPU algorithm might require some data structures which are persistent in the GPU memory (e.g., a precomputed read-only lookup table or intermediate results that are updated by multiple GPU tasks). We designate this case the *incremental case*.

Observations

We have implemented all the model cases in the most direct way and tested them in various combinations and settings using profiling techniques. The implementation used the OpenCL framework with out-of-order command queues and one main thread on the CPU. The main thread was used for dispatching commands to the GPU and control the data transfers. Following problems have been identified as the result of our experiments:

- There is a fragile balance between the CPU and GPU workloads. In many situations the CPU was waiting for the GPU and vice versa. The problem escalates significantly when multiple GPUs are employed in the system.

- The data transfers are especially problematic as they take considerable time and stall both the CPU and the GPU. The data transfers need to overlap with GPU computations, in order to reduce their effect on the performance.

- The data transfers are most efficient if aggregated in only a few bulk transactions. Unfortunately, the data gather operations that compact the input data in one block and the scatter operations that process the results of a GPU task take approximately the same time as the transfers themselves.

- Allocation and deallocation of the GPU memory is also bound with nontrivial overhead. It might be beneficial to reuse the allocated buffers, especially in the iterative case.

Multi-GPU Systems

We must also consider all model cases from the perspective of the multi-GPU systems. The simple case scenario is best suited for a single-GPU system; however, we can still benefit from having multiple GPUs if there are more subproblems to be solved by separate simple tasks. Otherwise, we can divide a simple GPU task into multiple tasks and use the iterative method to occupy more GPUs if the task is truly data parallel.

The iterative case scales almost ideally with the number of GPU devices available. We assume only that there are more GPU tasks than GPUs. If not, the size of the data blocks must be reduced so that more tasks are spawned. Finally, the incremental case has to use data replication so that the required data structures are copied to every GPU. This can be done only during the initialization stage in case of read-only lookup tables, or it must be performed on regular basis, if the data are mutable.

3.4.3 Framework Design

We have designed our framework as a flexible module which can be combined with various parallel libraries for multicore CPUs. The integration possibilities are described at the end of this section. The framework has two parts – a *GPU wrapper* that provides more suitable access to the OpenCL API with some additional features and a *feeding thread pool* of CPU threads. The overall design is depicted in Figure 3.8.

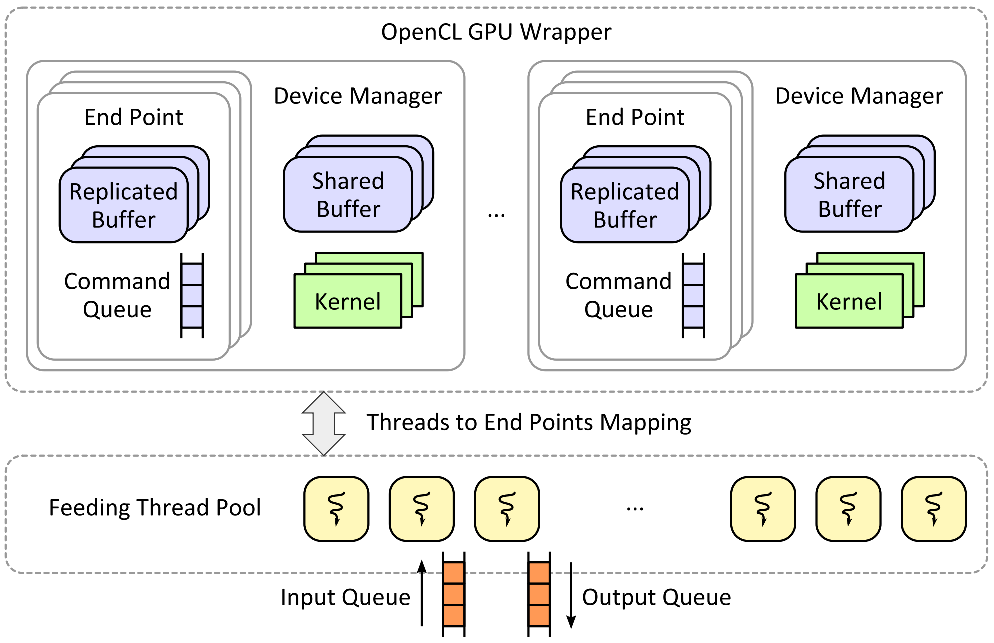


Figure 3.8: The design of the GPU task dispatching framework

The feeding thread pool plays a similar role to the backup threads. It handles asynchronous operations of the GPUs and related data transfer issues. The OpenCL wrapper was designed for several reasons. Most importantly, it automatizes certain routine operations (like kernel compilation) to reduce programmers work and it provides better means for binding the GPU devices with the feeding threads.

The GPU Wrapper

The wrapper is an object oriented API built on the top of the OpenCL runtime. It manages devices, memory buffers, and kernels. It is encapsulated in a singleton object, which is also a container for other objects. Each detected device in the system is handled by a *device manager* object which provides the API to work with the device and manages the necessary OpenCL structures (handles, context, etc.).

Each device manager is equiped with one or more *end points*. An end point is a logical structure that holds one command queue. Multiple end points may be attached to one device, so a concurrent execution of operations can be achieved – especially, overlapping the kernel execution and the data transfers.

The end points are also responsible for managing memory buffers. We recognize three types of buffers:

- *Anonymous buffers*, which are allocated for individual GPU tasks and that belong exclusively to a single end point. They are most suitable for the simple cases, when the buffer is used only once.
- *Replicated buffers* are allocated and registered under a name during the initialization phase. They are allocated through the device manager, which ensures that each end point allocates its own buffer. Hence, the GPU tasks can use any end point with the same functionality. These buffers are designed especially for the iterative cases.
- *Shared buffers* are similar to the replicated buffers, but only one instance is allocated by the device manager and it is shared by the end points. They are designed for the incremental cases, or as read-only data buffers. Note that the shared buffers do not ensure data replication/sharing among multiple devices. Every iterative case we have examined so far uses a different approach to replication (if any) and we have not found any uniform technique that would cover at least some of these situations both effectively and efficiently.

The wrapper also provides some additional support for kernels. The kernel compilation is rather runtime work, and it can get quite tedious if the programmer correctly checks all the API calls. A support for loading kernels from external text files was also added, so the kernel code can be outsourced to separate files.

The OpenCL API addresses kernel arguments by their position in the kernel header declaration. The wrapper extends this possibility by attaching names to the arguments, so they can be found more easily. It also automatically assigns named (replicated and shared) buffers to the kernel arguments of the same name.

The Feeding Thread Pool

The *feeding threads* were introduced to the framework in order to reduce the amount of time the GPU spends waiting for the input data from the CPU. They are handling the CPU part of the GPU task, which usually consists of gather/scatter routines and the host-device data transfers. The tasks are dispatched to the pool by one input queue. Completed tasks are stored into an output queue, which can be accessed by the remaining parts of the system, usually the main thread of the application. Both queues are thread-safe and blocking, thus they easily synchronizes the feeding threads. We did not use more elaborate techniques like task stealing for several reasons:

- This design is much simpler for implementation and for the programmer.
- The GPU tasks do not spawn another tasks, hence all tasks are issued from outside of the framework.
- The locking overhead of the queue is negligible in comparison to the execution time of the GPU tasks.
- The scalability is not an important issue since the number of feeding threads is proportional to the number of GPU devices and it is unlikely we will be able to squeeze more than 8 GPU devices into the conventional server computers.

The mapping between the feeding threads and the GPU end points is not hardwired in the framework. We can choose different approaches for different applications. One end point is always mapped to one feeding thread, but each thread may administer an arbitrary number of end points. The feeding thread dispatches incoming GPU tasks to associated end points in the round-robin style. In common cases, we have used the *thread-per-end-point* and the *thread-per-device* mappings.

The concept of a feeding pool was designed under the assumption that there are always more CPU cores than feeding threads. We have observed that there is no reason to have more than two end points per a GPU in normal situations, thus a four-GPU system requires at most 8 feeding threads. Mainstream CPUs have 12 logical cores at present time and multiprocessor NUMA servers are quite common. For these reasons we advocate that our assumption holds for the current mainstream hardware.

Integrating with Other Parallel Libraries

If the majority of the work is performed on the GPU, our framework can be used with only single main thread. The main thread may influence the scheduling up to a certain level by issuing the tasks to the input queue and waiting for them to complete. This way it may control the dependencies between the tasks or their priorities.

Since the input and output queues are thread safe, the GPU scheduler may be used in combination with any other parallel framework such as TBB, OpenMP, or Bobox. Parallel frameworks usually create as many threads as there are CPU cores available. Such situation is not optimal, because when the worker thread pool is combined with the feeding thread pool, the system ends with more threads than CPU cores. The operating system has to schedule these threads as evenly as possible, so the feeding threads get delayed by the workers and the GPUs get stalled.

Some of the libraries allow to configure the number of working threads, even to change it dynamically during the execution. If so, we can modify the size of the worker thread pool based on the number of the feeding threads, or even dynamically based on the number of the feeding threads that are currently busy.

We have considered combining the GPU framework with the Bobox system using the same attach/detach mechanism since the GPU tasks are in fact blocking operations from the CPU point of view. Unfortunately, the GPU tasks are rather complex and the OpenCL library uses a quite significant amount of CPU time which we cannot control. Every time we tried to decrease the priority or the number of the feeding threads, the overall performance had reduced significantly. It is our conclusion that the feeding pool must be treated with the highest priority and any other thread pool should utilize the CPUs only when they are not required by the feeding pool.

Our current implementation of the GPU scheduler operates with one input and one output task queue. It would be trivial to modify it to use multiple input/output queue pairs, so that multiple independent parts of the system may easily interoperate with our GPU scheduler. We are also exploring other possibilities of integration with the Bobox system and additional experiments are planned for future work.

3.5 Experimental Evaluation

The experiments focus on the empirical evaluation of two major contributions proposed in this chapter. After introducing the technical details, such as our hardware and methodology of measurement, we present the experiments proving that the attach/detach principle proposed for blocking operations in the Bobox system has a positive impact on the performance. The second half of the experimental section is dedicated to the performance evaluation of our GPU framework for hybrid scheduling.

3.5.1 Hardware and Methodology

The following experiments are oriented on performance, so the system real-time clock was used to measure the time required to complete each test. We realize that these times are strongly dependant on the hardware, the compiler used, and the implementation details. However, we have tried to maintain the same conditions for all related tests and we are mainly interested in the relative speedup rather than absolute time values. In case of the GPU framework, we have also measured GPU occupancy using the NVIDIA GPU profiler. The profiler times were compared with real times to verify that the profiler did not taint the results significantly.

The first set of Bobox experiments were performed on a Dell server with two Xeon E5310 processors, four physical cores running at 1.6 GHz each. The server was equipped with 8 GB RAM and two local 73 GB HDDs (spinning at $15,000$ rpm) connected in RAID 1. Red Hat Enterprise Linux 6.3 was used as the operating system on the server.

To demonstrate certain aspects of HDD controllers and I/O planning in operating systems, the second set of tests was performed on commodity personal computer. The PC was equiped with Intel Core i7 870 CPU running at 2.93 GHz, 8 GB of RAM and WD Raptor hard drive spinning at $10,000$ rpm. A 64 bit version of Windows 7 Professional operating system was used.

The GPU experiments were performed on a server built on a special motherboard (FT72-B7015) designed to embrace up to 8 GPU cards. The server was equipped with Xeon E5645 processor comprising 6 physical (12 logical) cores running at 2.4 GHz, 96 GB of DDR3-1333 RAM, and 4 NVIDIA Tesla M2090 GPU cards based on the Fermi architecture. Each GPU chip consists of 512 cores (32 cores per 16 SMPs) and 6 GB of memory.

We also tested the GPU implementation on a commodity PC with two gaming cards NVIDIA GTX 580. These cards have also 512 cores, but only 1.5 GB of memory. We have found that the GTX 580 cards have similar performance as the Teslas, thus we do not provide more detailed comparison.

3.5.2 Blocking Tasks

As we have mentioned in Section 3.3.2, one of the most important differences between the Bobox system and other libraries is the support of blocking operations. Both experiments in this section focus on them.

Blocking Operations

The first experiment compares how the TBB library and the Bobox deal with the blocking I/O operations in the tasks. The experiment comprised 40 tasks executed concurrently with no dependencies. Each task generated 32 million of 32 bit values using 100 iterations of *linear congruential generator* [73] and then it wrote the entire 128 MB block into a binary file. The file was flushed after the write to ensure the write operation would take place immediately. In the serial settings, the random generator took approximately 685 seconds of CPU time while the writing operations took 132 seconds. Hence, the computations took 84% of total (serial) time and the writing took 16%.

The TBB experiment used the `parallel_for` template to execute the requests in parallel. The Bobox framework was tested under multiple settings. A B_i setting denotes a Bobox experiment, that used i backup threads. The first setting B_0 does not use backup threads at all, thus the invocations of the `detach` and `attach` functions have no effect. Both the TBB and the Bobox used as many computational threads as there were available CPU cores.

The results of this experiment are summarized in Figure 3.9. They clearly prove that the backup threads help the overall performance significantly. We have also tried different numbers of tasks, different ratio of computations and I/O, and the measured improvement of the backup threads was similar in every case.

Hard Drive Limitations

The first experiment proved, that the backup threads improve the performance of the blocking tasks. However, most of the blocking tasks (like I/O) are performed

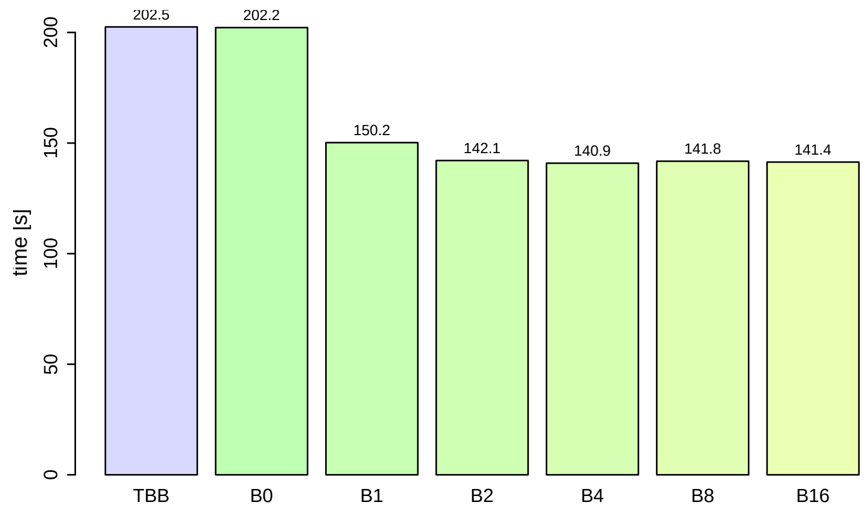


Figure 3.9: Execution times (in seconds) of 40 tasks with a blocking operation

on devices, which cannot handle many concurrent operations. To test the limits of this approach, we have designed another experiment.

The second experiment had also executed 40 tasks, but it used only one iteration to generate the data. The computational work was reduced 100×, thus the most of the time was taken by the writing operations. The test was conducted on a commodity PC with a single hard drive, which does not handle concurrent operations as well as the RAID controller of the server.

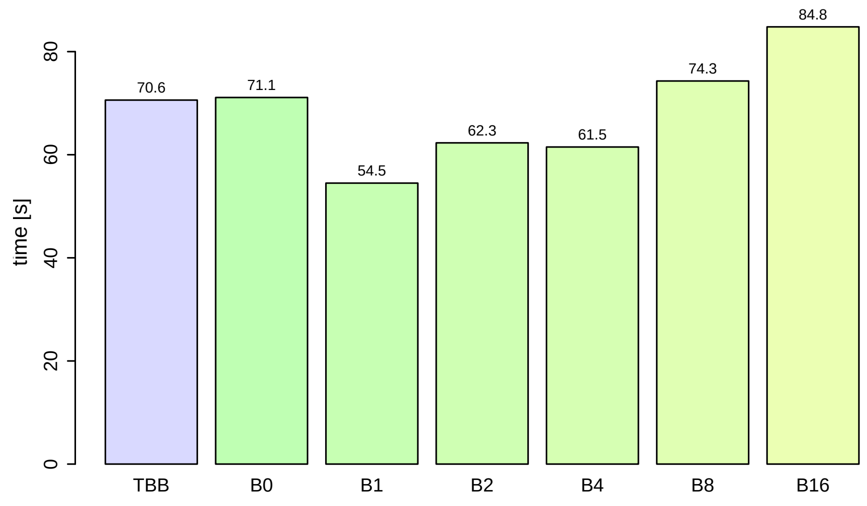


Figure 3.10: Execution times (in seconds) of tasks comprised mostly of writing

We used the same configuration for the TBB and the Bobox as in the previous experiment. The results are presented in Figure 3.10. They indicate, that any attempt for concurrent writing was slower than the sequential writing. This confirms our original assumption that some hard drive controllers (especially those in common personal computers) cannot handle the parallel workload as well as RAID controllers in servers. Therefore, the best performance was achieved by the Bobox system with one backup thread that serializes the I/O operations but also leaves the worker threads free for computations.

Use of the `detach` and the `attach` functions improved the performance significantly. However, the proper size of the backup thread pool must be determined for each type of blocking tasks (or each device). We will focus on this area in our future research.

3.5.3 Hybrid Scheduling

To present the benefits of our GPU scheduler, we chose an image similarity search problem. This problem and its GPU solution is thoroughly described in Chapter 4. We summarize it briefly just for the purposes of these tests. The problem of the similarity search is based on the query-by-example paradigm. Let us have a database of images that are not annotated or otherwise classified. The user cannot search such database using a conventional text-query interface, but rather provide an example image and expects to get similar images in response.

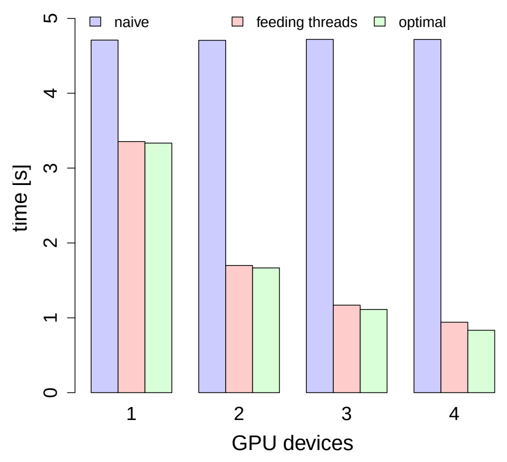
The images are represented as signatures, each signature is a set of points in a 7-dimensional space with weights. A metric distance function (Signature Quadratic Form Distance in our case) is defined to compute the distance (inversed similarity) of image signatures. The distance between a query signature and all (or at least a subset of) the signatures in the database needs to be computed in order to determine the results of the query. The distances are computed iteratively on available GPUs and each GPU task has the following steps:

1. Gather image signatures into a single block.
2. Copy the signatures to the GPU in one transfer.
3. Invoke the SQFD kernel that computes distances from the query to all signatures in the block.
4. Transfer the distances back to the host memory.
5. Use the distances to determine which images from the block will be included into the result.

In the following experiments we compare our solution to the original naive approach which uses a single CPU thread to both dispatch the work to the GPUs and process the distances to create the k nearest neighbour result. The scheduler uses 2 end points per GPU[5] and one feeding thread per end point. We provide results measured for different numbers of GPU cards to determine the scalability of the solution.

Figure 3.11 depicts the results of the experiments. The left graph shows absolute times required for processing our test database comprising $950,000$ images.

[5]This was empirically determined as an optimum for overlapping data transfers and computations.



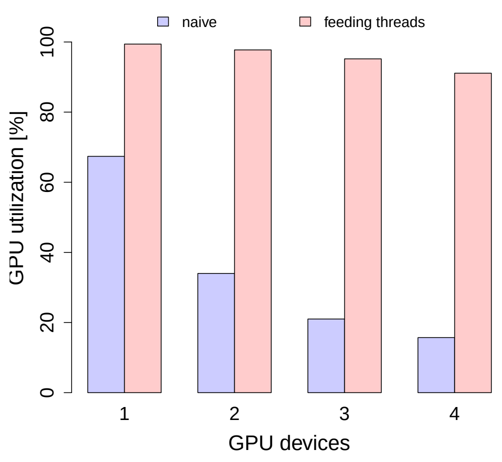


Figure 3.11: Measured real times in ms (left) and average utilization of the GPUs in percents (right)

As we can see, the feeding threads helped significantly in achieving better performance (1.4× for single GPU and 5× for 4 GPUs) and the problem scales almost idealy with increasing number of GPUs. The data marked *optimal* represent the theoretical peak performance of our algorithm, that would be achieved, if all the GPUs were utilized for 100%.

The reason for this behaviour is visualized in the right graph of the figure. It shows the utilization of the GPUs in percents of the total computational time. Our scheduler is capable to utilize the GPUs for more than 90%, even when 4 GPUs are being served. We believe that the GPUs are not fully utilized because of the throughput limits of the PCI-Express and memory buses. The naïve approach utilizes a single GPU only up to 65%, and adding more GPUs does not improve the situation since their utilization decreases in the proportion of their numbers. Furthermore, we have observed that our scheduler overlaps data transfers with SQFD computations significantly, while the naive approach does not.

4. Accelerating Similarity Search

In the following two chapters, we shift our focus to multimedia databases, which are standing on quite different paradigms than the relational databases. This domain presents even more computationally challenging tasks, thus it might benefit from the parallel acceleration even more. We have done an extensive research in the area of image databases and content-based retrieval based on the query by example model.

4.1 Introduction

Searching textual data on a web scale has been attended by the largest companies in the IT industry (like Google, Microsoft, or Yahoo) and they have perfected the text-based search significantly in the last two decades. However, when dealing with multimedia content, there is still much room for improvement. In the past, the major approach was also text-based. This approach expects the multimedia content is annotated by textual meta information, such as captions, labels, tags, comments, ratings, or a surrounding content gathered from the adjacent web page.

As the multimedia content grows almost proportionally to the current storing capacities, the users are becoming more and more reluctant to provide any textual annotation to their multimedia as it consumes a lot of time. In the direct response to this trend, new techniques of querying data have been established. These techniques might be more convenient for the user, but they also require much higher computational power. This approach is called the *content-based retrieval* as it relies on the content analysis and comparison in the search process. It is described in more detail in Section 4.1.1.

The similarity search principle brings two major challenges: How to represent the content so it can be easily queried and how to measure the similarity between the two objects. The objects are usually represented by some kind of *descriptors*. Several examples of descriptors, especially for the image content, are presented in Section 4.1.2. The descriptors are then measured with a *distance function*. The distance function is defined as the opposite of similarity, so it is sometimes also called the *dissimilarity function*. We address the issue of similarity functions in Section 4.1.3.

There are also some other, more philosophical, issues regarding the similarity search. Every query-based retrieval system struggles with the query interpretation problem. The user must formulate his/her requests into a query and the system must understand the query correctly. This might be difficult, as the query may not reflect the intent of the user properly, it may be provided within a user-specific context, or it could be ambiguous due to the imperfections of the query language. These issues are quite serious in the text-based searching and they get even worse in the similarity search. However, we do not address these issues as our main objective is to improve performance of the current methods by employing many-core GPUs.

4.1.1 Content-based Retrieval

The content-based retrieval paradigm was introduced to improve the search process in the databases, where textual annotations are unavailable, incomplete, difficult to acquire, or even impossible to establish. This reflect the situation in databases of multimedia content (images, audio, or video) [74], bio-information databases (such as protein structures), or other highly specialized and structured objects.

The content-based approach is based on the idea, that the database should use the content of these complex objects and provide content-specific means to query the data.

The Query

The most basic query principle in the similarity search is the *query by example*. The user provides an example of an object and expects to receive the same or very similar objects in return. This method is very intuitive and easy to use. On the other hand, the user must provide an example of the object. Such example may be hard or even impossible to get and it may not express exactly what the user wants.

Another method is the *query by description*. The user describes the object in well-established terms or even in the natural language. This may be more convenient in case the user cannot provide an example for the previous method. On the other hand, it tends to have very bad results as the textual description of the object is very inaccurate in most domains.

Somewhere between the first two approaches would be the *query by sketch*. It is basically very similar to the query by example, but the user does not need to provide an entire example. The query is represented by only a *sketch* of an object, which could be a quick hand drawing as a query for the image database or a whistle of a melody as a query for the musical database. This might be a good compromise in some cases, but it might require additional skills from the user, such as talent for hand drawing or the ability to whistle in the tune.

In the following work, we prefer to use the query by example model, as it has proved quite useful for search in an image database. Digital cameras are a quite common equipment of laptops and cell phones, thus the user should not have much trouble acquiring queries from the real world. Furthermore, if the user is not able to provide the query, it can be crossed with the text-based retrieval methods and employed in an explorational model [75, 76].

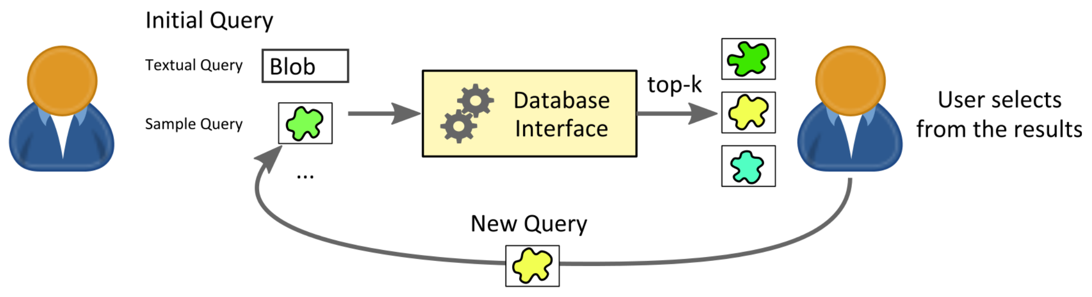


Figure 4.1: The explorational approach to querying image databases

The Figure 4.1 depicts the exploration process for the image database. The user initiates the first search by the means of the text-based search (or by some other method) and then continues interactively by exploring the database. In each following step, the user explores the results of the previous step, selects an image (or even multiple images) that resembles what the user is looking for the most and uses this image as a query for a new search. Doing so, every iteration brings the user closer to the desired results.

The Selectivity of Queries

One of the most common types of queries is the k *Nearest Neighbours* query (kNN), also known as the top-k query. The search returns exactly k results, which are the best in terms of the search. Using the distance (dissimilarity) function d, a kNN result set $R_{kNN}(q) \subset D$ of a query q represents the k closest objects selected from the database D, that means $|R_{kNN}(q)| = k$ and $\forall x \in R_{kNN}(q), \forall y \in D \setminus R_{kNN}(q) : d(q,x) \leq d(q,y)$. This type of queries is directly applicable in the user interface as it represents, what the user intuitively expects.

The second type of content-based queries is the *range query*. It is parametrized by a range value r and the result $R_r(q)$ comprise all objects which are closer to the query than this range (i.e., $\forall x \in R_r(q) : d(q,x) \leq r$ and $\forall y \in D \setminus R_r(q) : d(q,y) > r$). Range queries are used in more specific situations when the range of the query can be provided (e.g., by some estimation mechanism or as a result of a previous query). Their direct usage is complicated by the fact, that unlike the k value from the kNN query, the r parameter does not have any intuitive meaning as the distances usually capture only a relative dissimilarity. The kNN query can be sometimes perceived as a special type of the range query, where the r is equal to the distance of the k-th item. However, this point of view is purely theoretical since the value of r is not known before the query finishes.

For the purpose of an easy reference, we define the *filtering range* of a query as follows: in case of range queries, the filtering range is constant and equal to r during the whole query processing. The kNN queries filtering range decreases dynamically and it is equal to the maximal object distance in the result set R (i.e., $\max\{d(q,o)|o \in R\}$), or to the infinity when $|R| < k$. The filtering range is used to decide, whether a candidate object is about to be included into the result.

Query Evaluation Algorithms

For a better understanding of the principles of kNN and range queries, we present the algorithms that resolve these queries. These simplest algorithms are sometimes designated the *sequential scan* as they read the database objects sequentially, compute their distances to the query, and update the result set. Algorithm 4.1 represents a version of sequential scan for range queries and Algorithm 4.2 is a version for kNN queries.

```
**Data**: database (signatures) $D$, query signature $q$, range $r$
**Result**: list $R$ of objects within the given range (and their distances)
$R \leftarrow \emptyset$
**foreach** $s \in D$ **do**
    **if** $d(q, s) \leq r$ **then**
        add $s$ to $R$
    **end**
**end**
```

Algorithm 4.1: Sequential scan for range query

The kNN query algorithm is only slightly more complicated. The result set R has limited capacity of k and we require to find the maximal distance of the objects in R. We can represent the R by a 2-regular heap data structure which keeps inserted objects and their respective computed distances. The heap has $\Theta(1)$ time complexity for accessing the maximum and $\mathcal{O}(\log k)$ for adding and removing items.

```
**Data**: database (signatures) $D$, query signature $q$, parameter $k$
**Result**: list $R$ of the $k$ closest objects (and their distances)
$R \leftarrow \emptyset$
**foreach** $s \in D$ **do**
    **if** $|R| < k \vee \max\{d(q, o)|o \in R\} > d(q, s)$ **then**
        add $s$ to $R$
        **if** $|R| > k$ **then**
            select $x \in R : d(q, x) = \max\{d(q, o)|o \in R\}$
            remove $x$ from $R$
        **end**
    **end**
**end**
```

Algorithm 4.2: Sequential scan for kNN query

The distance functions are usually rather expensive to compute. Even though the function d is used multiple times in the algorithm, we would like to emphasize that the distance to each database object is computed exactly once and already computed distances are cached in R.

4.1.2 Object Descriptors

Usually, it is quite impractical to compare the database objects directly. Multimedia objects contain many information, some of which may not be relevant for the similarity measure. In order to simplify the distance function and reduce the amount of data required for the computation, the objects are represented by descriptors.

A *descriptor* is a footprint of an object that aggregates features which are relevant for the similarity measure. In case of images, the descriptor can extract some color information, texture information, or detect some kind of concepts in images. Musical database can analyze rhythm, frequencies, or detect instruments for the descriptors.

The descriptors are an inseparable part of the similarity model and they are tightly entangled with the distance function. They can affect the efficiency in the terms of time and space complexity as well as the quality of the similarity model. Since we focus on the image data, we present some examples of descriptors used for images. At the end of this section, we describe the feature signatures used as descriptors in our similarity model.

Image Descriptors

Image descriptors try to capture various image properties, such as color or texture. More complex signatures attempt to detect some points of interest or even some concepts in the image. One of the simplest descriptors is a color histogram. Histograms usually have fixed size and comparing two histograms can be quite fast. On the other hand, color histogram dismisses the spatial distribution of the colors and it does not provide very precise results [77].

A classification approach to the problem takes the *bag-of-words* model [78]. It is inspired by document indexing techniques and it treats image features as words. A *vocabulary* of features is constructed form the images in the database and then the images are represented by a sparse vector of occurrence counts of words from the vocabulary.

The bag-of-words model can use various types of features to create the vocabulary. One of the most famous is the Scale-invariant feature transform (SIFT) [79]. These features represent interesting points in the image (*keypoints*), which characterize objects. These points should be detectable, even if there is a change in the image scale, noise, or illumination. Therefore, the extraction process usually relies on edge detection algorithms.

Another approach is to sample the image features and to create feature signatures. Image features are localized, thus each feature is provided with image coordinates (x, y), that correspond to a location from which the features has been extracted. The feature vector usually contains color information taken from a pixel at x, y, or the average color of a small surroundings of x, y. It may also contain some texture information like contrast or entropy, some information about detected edges, or color gradients. We use this type of descriptors in our work so we revise it later in more detail.

Much broader, but also more complex, approach was taken by the MPEG-7 descriptors [80]. MPEG-7 is a set of ISO standards designed for multimedia content description. It uses XML to store metadata along with the multimedia content. These standards are quite extensive and they define the description language, query formats, and various descriptor types for images, audio, video, and even 3D objects. The standard for visual descriptors covers colors and their distributions, textures, illumination, edges, shapes, and even face recognition.

Image Feature Signatures

As mentioned before, we employ *image signatures* in our similarity model. It is a simple indexing technique, which allows us to represent images in a more compact way that is also more suitable for a fast similarity comparison. An image signature gathers a specific feature information about the image parts, such as color or texture entropy. We use 7 dimensional feature space (x, y, L, a, b, c, e), where x, y are normalized coordinates of the feature position in the image, L, a, b is the color information converted into a Lab space [81], c is the *contrast* value and e is the *entropy*.

Many features are sampled from the image producing many points in the 7-dimensional feature space. A clustering algorithm is applied to aggregate this information into more compact form. The image signature is then represented by the cluster centers (*centroids*) and a weight of the cluster, which is computed from the number of points belonging into that cluster. The Figure 4.2 depicts an example of images and simplified visualisation of their corresponding feature signatures.

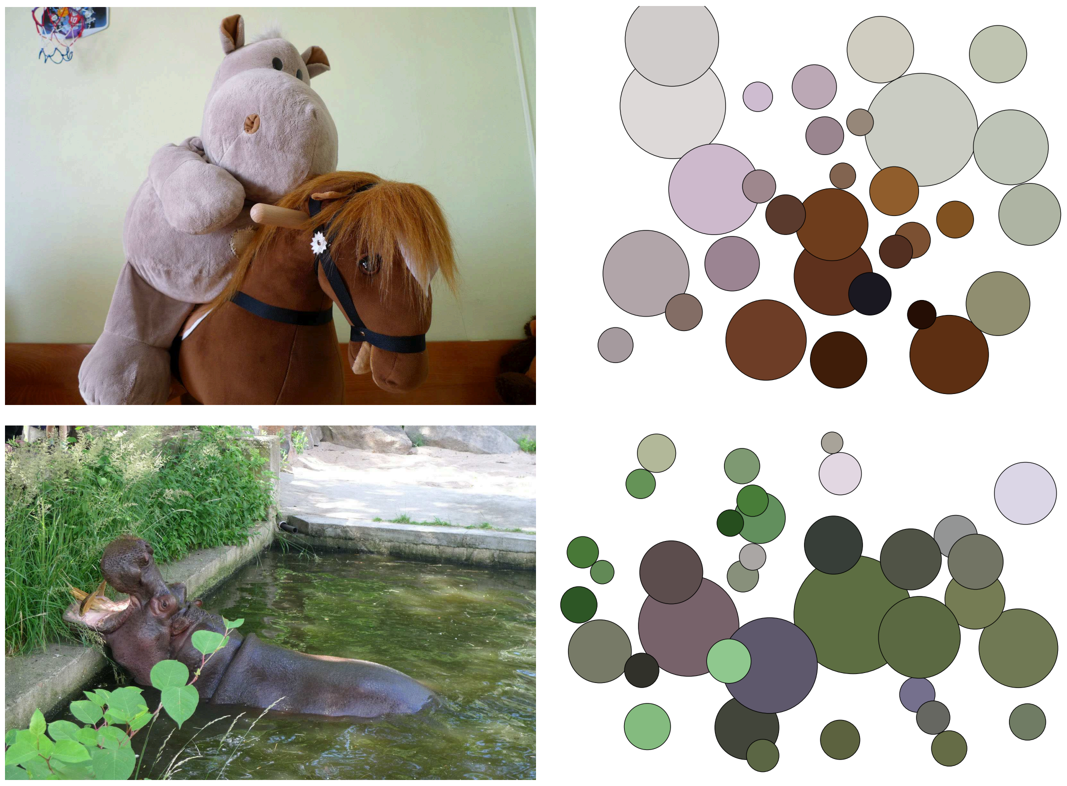


Figure 4.2: Example of images and their signatures

More detailed description of the image signatures and their extraction is provided in Chapter 5. For the purposes of the similarity search, it suffices to know that a signature is a set of 7-dimensional points with weights.

Formally, a signature S^o of an object o is defined as $S^o = \{(c_i^o, w_i^o) | i = 1 \ldots n\}$. The $c_i^o \in \mathbb{R}^7$ and $w_i^o \in \mathbb{R}^+$. We have to emphasize that the number of centroids differs for each object as simpler objects are covered by fewer centroids while complex images require more centroids to capture the same level of detail. We denote the number of centroids (i.e., the size of the signature) $|S^o|$.

4.1.3 Distance Functions

A distance function, also designated as a dissimilarity function, is the nemesis of object descriptors in a similarity model. It computes the inverse value of similarity between two objects.

The most common family of distance functions, which are applied in simple cases or as ground distances for more complex measures, is the L_p metrics. If the descriptor comprise a vector of fixed length d, it can be perceived as a point in d dimensional space, $\mathbb{R}^d$ for example. The L_p distance between points x and y in such space is defined as $L_p = (\sum_{i=1}^{d} |x_i - y_i|^p)^{1/p}$ L_1 (the *Manhattan distance*) and L_2 (the *Euclidean distance*) metrics are used the most often.

Adaptative Measures

When the descriptors have variable length, an adaptative measure has to be used. The descriptor can be organized as a simple set of properties, or it can also capture the ordering, so the properties are stored in a sequence. We present an example of a measure for each case.

One of the simplest similarity measures for sets is the *Jaccard similarity coefficient* or the *Jaccard index*. This coefficient is defined as a ratio of the number of properties common for both sets to the number of unique properties in both sets. Formally, for sets X and Y, the Jaccard index is

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}.$$

In addition, the *Jaccard distance*, which measures the dissimilarity between two sets is defined as $J_\delta(X,Y) = 1 - J(X,Y)$. This concept can be extended by altering the definition of set union and intersection for the descriptor sets or by choosing a different way of computing their cardinalities.

In case of sequential descriptors, the measure must respect the ordering and project it to the distance. A typical example of such measure is the *Levenshtein distance*. It was originally designed as an edit distance between two strings, but it can be used in a similarity search to compare sequences. It specifies three basic editing operations:

- inserting a character,
- deleting a character,
- and replacing a character.

A distance between two strings is then defined as the smallest amount of basic editing operations required to transcribe one string to the other. The distance can be efficiently computed using dynamic programming. Time complexity of the algorithm is $\Theta(m \cdot n)$, where m and n are the lengths of compared strings.

Measures for Feature Signatures

As mentioned before, we use the image feature signatures as object descriptors in our similarity model. The signatures have variable length and the features are weighted, thus we require adaptative measures that can deal with weights. There are a few such measures and we introduce three of them here.

The *Hausdorff metric* [82] is designed to measure distance between two subsets of a metric space. Feature signatures are in fact sets of points from $\mathbb{R}^7$. The measure is defined as the maximum of individual distances between each point of one set and a corresponding closest point in the other set. The formula can be formalized as follows. Let us have two subsets X, Y of some metric space (M, d). The Hausdorff distance is then defined as

$$d_H(X,Y) = \max\{\sup_{x\in X}\inf_{y\in Y} d(x,y), \sup_{y\in Y}\inf_{x\in X} d(y,x)\}.$$

As we operate on finite sets, the supremum and infimum in the formula can be replaced with maximum and minimum respectively. If the feature signatures did not have the weights, the distance d between two features could have been a simple L_p metric, euclidean L_2 for instance. The weights can be incorporated in many ways. One of them is to multiply the individual distances by the weights of adjacent points.

The greatest problem of the Hausdorff distance is that it takes the maximum of the individual distances. Therefore, one outliner among the features can cause that two very similar objects are measured as quite distant. Better results provides the *Signature Quadratic Form Distance* (SQFD) [83]. It is based on a similarity function f_s that measures similarity between two features. The quadratic form is created by enumerating all f_s values for all feature pairs. We describe this function thoroughly in Section 4.1.4 as we use it in our similarity model.

The SQFD has a specific variation called GQFD [84], which is based on the Gaussian mixture models instead of plain signatures. This model represents the features as probabilistic functions instead of fixed points in the feature space. The SQFD can be perceived as a special case of GQFD with Dirac delta function used for the probability distribution.

Another example of a distance function is the *Earth Mover's Distance* (EMD) [85], which can be used to compare not only signatures, but also histograms and other types of descriptors. The distance is defined as the cost of transforming one feature signature to another. It can be considered a transportation problem, thus it can be solved by linear optimization methods, such as specialized Simplex algorithm.

Given a ground distance d, which measures dissimilarity of two features, the EMD is defined as a minimum cost flow over all flows $f_{ij} \in \mathbb{R}$:

$$d_{EMD}(S^q, S^o) = \min_{f_{ij}} \left\{ \frac{\sum_i \sum_j f_{ij} \cdot d(c_i^q, c_j^o)}{\min\left\{\sum_i w_i^q, \sum_j w_j^o\right\}} \right\},$$

The distance is subjected to the constraints: $\forall i : \sum_j f_{ij} \leq w_i^q$, $\forall j : \sum_i fij \leq w_j^o$, $\forall i,j : f_{ij} \geq 0$, and $\sum_i \sum_j f_{ij} = \min\{\sum_i w_i^q, \sum_j w_j^o\}$. These constraints guarantee a feasible solution. All costs must be positive and limited by their weights. The ground distance d could be an L_p metric for instance.

The EMD function usually performs quite well in the terms of similarity model precision. However, its computational cost is rather high. The linear optimization methods applied for this problem are known to have $\mathcal{O}(n^4)$ time complexity.

4.1.4 Signature Quadratic Form Distance

The *Signature Quadratic Form Distance* (SQFD) [83, 86, 87] is adaptative similarity measure for multimedia signatures. We have used the SQFD in our experiments since it provide very good similarity precision, it is easy to implement, and it is quite fast to compute [77]. It is based on the classical quadratic form distance, which can be used to compare histograms [88] for instance. The classical QFD applied a cross-dimension concept to compare all the dimensions of the feature histogram. This method is adopted to a cross-dependency concept that compares all feature signatures with each other.

Mathematical Definition

We define the distance on the signatures definition described at the end of Section 4.1.2. The compared signatures S^q and S^o are representing the query and the object respectively. Let us revise, that signature is a set of features $S^o = \{(c_i^o, w_i^o) | i = 1 \dots n\}$, where each feature is represented by a centroid $c_i^o \in \mathbb{R}^7$ and weight $w_i^o \in \mathbb{R}^+$. The distance is formalized as

$$d_{SQFD_{f_s}}(S^q, S^o) = \sqrt{(w_q | - w_o) \cdot A_{f_s} \cdot (w_q | - w_o)^T}.$$

The vector $(w_q | - w_o)$ is created by concatenation of weight vectors w_q and $-w_o$, where $-w_o$ has negated values. The concatenation of $w_q = (w_1^q, \dots, w_n^q)$ and $w_o = (w_1^o, \dots, w_m^o)$ looks like $(w_q | - w_o) = (w_1^q, \dots, w_n^q, -w_1^o, \dots, -w_m^o)$. The values n and m are shorthand notations of the sizes $|S^q|$ and $|S^o|$ respectively.

The similarity matrix $A_{f_s} \in \mathbb{R}^{(n+m)\times(n+m)}$ is the enumeration of similarity function f_s applied to all pairs of centroids concatenated from both signatures. Let us denote $c = (c_q | c_o)$ the concatenation of c_q centroids and c_o centroids $(c_1^q, \dots, c_n^q, c_1^o, \dots, c_m^o)$. The elements of similarity matrix A_{f_s} are then defined as $a_{ij} = f_s(c_i, c_j)$, where $i, j = 1, \dots, n + m$. Since the matrix represents all pairs of the concatenated centroid vector c, it comprises the self-similarity parts representing internal similarity of the signatures as well as inter-similarity between the two signatures. The matrix structure is depicted in Figure 4.3.

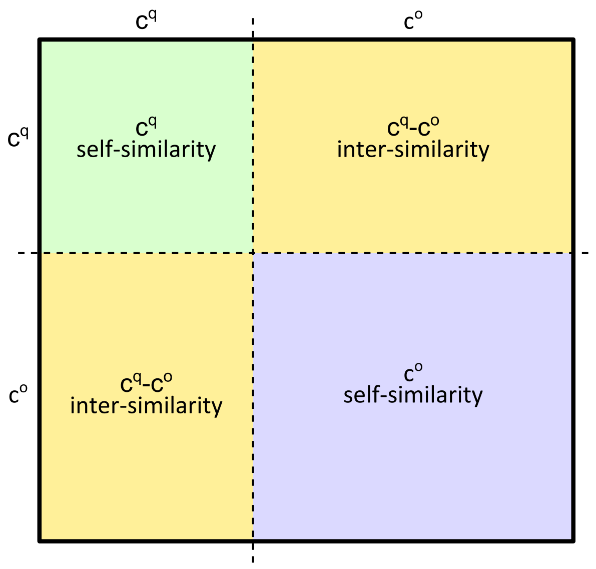


Figure 4.3: Similarity matrix A_{f_s} of the signatures S^q and S^o

Similarity Functions

The similarity function $f_s(c_i, c_j) \mapsto \mathbb{R}$ measures similarity between two centroids. An inequality $f_s(c_i, c_i) \geq f_s(c_j, c_k)$ must hold for all c_i, c_j, c_k, where $c_j \neq c_k$. The similarity function is expected to assign higher values to more similar objects and identical objects must be ranked more similar than any two nonidentical objects.

There are many possibilities, how to define similarity function. One is to base the similarity on *ground distance functions*, that are naturally introduced in the features space. In case of $\mathbb{R}^n$ spaces, a ground distance function could be L_p metric for instance. For given distance function $d(c_i, c_j) \mapsto \mathbb{R}^+$ and parameter α, there are three typical similarity functions:

- Minus function $f_-(c_i, c_j) = -d(c_i, c_j)$
- Gaussian function $f_g(c_i, c_j) = e^{-\alpha \cdot d^2(c_i, c_j)}$
- Heuristic function $f_h(c_i, c_j) = \frac{1}{\alpha + d(c_i, c_j)}$

In the following work, we have exclusively used the Gaussian function with L_2 Euclidean metric as the ground distance function $f_s(c_i, c_j) = e^{-\alpha \cdot L_2^2(c_i, c_j)}$. The parameter α is used to tune the ratio between precision of the similarity model and the *intrinsic dimensionality*, which affects the indexability of the database. The intrinsic dimensionality of a database is defined as $iDim = \bar{d}^2 / 2\sigma_d^2$, where $\bar{d}$ is the mean value and σ_d^2 is the variance of all distance values between objects in the database. Database with low intrinsic dimensionality can be effectively indexed by techniques described in Section 4.2. Databases with high intrinsic dimensionality are hard to index and we need to use a simple sequential scan to resolve queries.

Implementation and Optimization

Since SQFD is the key function implemented in our similarity search system, we provide a little more insight into implementation and optimization details for CPU systems. First of all, let us make an observation, that we do not need to keep entire A_{f_s} matrix in the memory. The first multiplication $(w_q| - w_o) \cdot A_{f_s}$ produces a vector of the same proportions as the $(w_q| - w_o)$. Therefore, the A_{f_s} values can be computed on the fly and partial sums can be maintained in the results vector. This intermediate vector is most likely to fit the L1 cache as we expect the signatures have hundreds of centroids at most. Finally, the computations of A_{f_s} values can be partially parallelized employing SIMD vector instructions like SSE.

If we look beyond the straightforward optimizations, we can remove some redundant computations. The Figure 4.3 shows that the similarity matrix is naturally divided into four parts. Two parts represent the self-similarities of the S^q and S^o signatures and the remaining two parts represent the inter-similarity between these signatures. If the similarity function f_s is symmetric, the inter-similarity parts are identical (only transposed) and the self-similarity submatrices are also symmetric. We can use that to compute only half of the A_{f_s} values.

In the traditional searching scheme, when the distances between a query and database objects are computed, more computations can be saved. The self-similarity matrix of the query can be computed only once at the beginning of the search. The self-similarity matrices of the database objects can be precomputed and stored along with the database. Furthermore, we can multiply these matrices by the corresponding parts of the weight vectors and keep them as a single number, thus in a very compact way.

4.2 Indexing

The basic algorithms for kNN and range queries expect, that a distance is computed between the query object and each object in the database. This approach does not scale for databases of the world wide web magnitude. The most work of the search process is usually spent by computing distances and loading the object descriptors. Indexing techniques are designed to reduce the number of distances being computed, thus the descriptors being loaded. This pruning method is sometimes called *prefiltering*. Prefiltering process yields object *candidates* that are worth further examination. Only the distances to the candidates are computed and *filtering* is used to determine, which candidates are finally inserted to the result set.

The indexing techniques can be divided into two basic categories: *metric* and *nonmetric*. Metric indexing (also called called *metric access methods*) [89, 90] applies for similarity models that create a metric space. In fact, many similarity models fall into this category, so we address these methods in more detail in the remaining of this section.

If the distance function of the similarity model does not conform with metric axioms, a nonmetric indexing methods [91] have to be used. In some cases, a domain expert can identify specific properties of the similarity model, which can be used for indexing. Otherwise a general nonmetric access method must be used.

The most of nonmetric access methods are based on some kind of mapping. A projection is created from the original (nonmetric) space into a metric space, vector space, or even Euclidean space. The projection must respect some properties of the target space, so that indexing techniques for metric, vector, or Euclidean space can be used. Some more elaborated structures like NM-tree [92] are based on metric indexing methods, but they use some form of mapping to modify the method for nonmetric spaces. A thorough comparison of nonmetric methods is presented in the work of Skopal and Bustos [91].

4.2.1 Metric Spaces

Properties of the metric space can be used for object pruning, which inherently improves the search efficiency. A metric space is a ordered pair $(\mathcal{M}, d)$, where $\mathcal{M}$ is a set and d is a metric distance function $(d(x, y) \mapsto \mathbb{R}, x, y \in \mathcal{M})$ that measures distance between objects from the set $\mathcal{M}$. Additionally, the metric function complies with the metric axioms:

1. $d(x, y) \geq 0$ (nonnegativity)
2. $d(x, y) = 0,$ iff $x = y$ (identity)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, y) + d(y, z) \geq d(x, z)$ (triangular inequality)

The most important is the fourth axiom, the triangular inequality. It can help the searching process to rule out some of the database objects without computing the (potentially expensive) distance function. Let us have a query object q and two database objects p, o. The triangular inequality can be used to determine a *lower bound estimate* of the distance $d(q, o)$ from the distances $d(q, p)$ and $d(p, o)$. The inequality and the estimate are visualized in Figure 4.4.

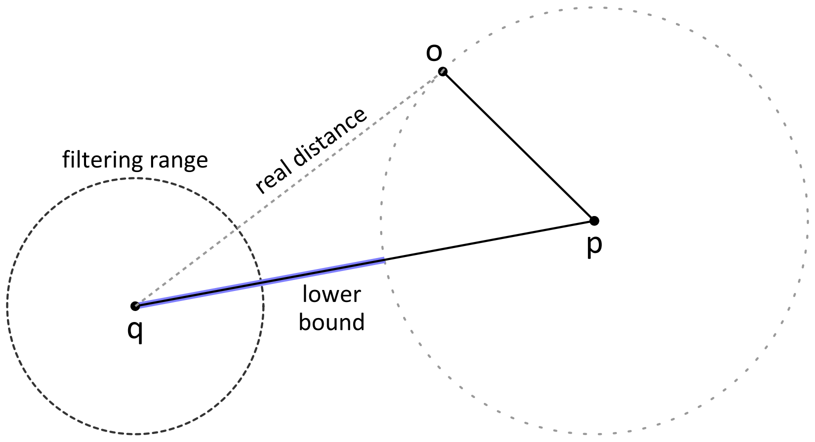


Figure 4.4: Lower bound estimation by triangular inequality and one pivot

The triangular inequality states that $d(q, o) + d(o, p) \geq d(q, p)$, which also means that $d(q, o) \geq |d(q, p) - d(o, p)|$. We can use this formula to define the lower bound estimate of the real distance, as $lb_p(q, o) = |d(p, q) - d(o, p)|$. Since the estimate is called *lower* bound, it is always smaller or equal to the real distance ($lb_p(q, o) \leq d(q, o)$).

More objects like p may be used to estimate a better lower bound. For a set of objects $P \subseteq D$ a combined lower bound estimate $lb_P(q, o)$ is defined as the most accurate lower bound $lb_{p_i}(q, o)$ of all $p_i \in P$. The estimate is more accurate if it is closer to the real distance, thus greater. Hence, the $lb_P(q, o) = \max\{lb_{p_i}(q, o)|p_i \in P\}$. If the query object q or the set of prefiltering objects P is predefined and fixed, we use a shorthand notation $lb(q, o)$, $lb_P(o)$, or even $lb(o)$ for the lower bound $lb_P(q, o)$.

4.2.2 Pivots

Objects used for estimating the lower bounds are usually called *pivots* or *vantage points*. We always denote the pivot set $P \subseteq D$, where D is the database unless specified otherwise. There are various methods that use pivots for prefiltering [89]. The basic idea, which is common to all of them is that the distances between pivots and database objects are precomputed, thus they do not need to be computed when the query is being resolved. We denote $d_P(p, o)$ the distance between pivot p and object o which has been precomputed ($d_P(p, o) \equiv d(p, o)$) to distinguish between precomputed values and distances computed during the query evaluation.

Approximating Eliminating Search Algorithm

One of the first methods based on this approach was the *approximating eliminating search algorithm* (AESA) [93]. It expects that the index is formed of a matrix, that holds precomputed distances between every object pair in the database. In another words, the pivot set comprises entire database ($P = D$). Every time a distance is computed between one of the objects, the object is used as pivot for prefiltering the candidates. The pseudocode of the kNN query with AESA access method is presented in Algorithm 4.3.

```
Data: database $D$ ($\forall o_i, o_j \in D$ precomputed $d_P(o_i, o_j)$), query $q$, $k$
Result: list $R$ of the $k$ closest objects (and their distances)
$C \leftarrow D, R \leftarrow \emptyset$
$\forall o \in D : lb(o) \leftarrow 0$                     // initialize $lb$ estimates
while $C \neq \emptyset$ do
    $c \leftarrow$ object from $C$, $C \leftarrow C \setminus \{c\}$
    // update the result set
    if $|R| < k \vee d(q,c) < \max\{d(q,o)|o \in R\}$ then
        $R \leftarrow R \cup c$
        if $|R| > k$ then
            remove object $o$ with the greatest $d(q,o)$ from $R$
        end
    end
    foreach $o \in C$ do                              // update $lb$ estimates
        $lb(o) \leftarrow \max\{|d_P(c,o) - d(c,q)|, lb(o)\}$     // triangular ineq.
    end
    if $|R| \geq k$ then
        $r \leftarrow \max\{d(q,o)|o \in R\}$                  // filtering range
        foreach $o \in C$ do        // filter $C$ to prune distant objects
            if $lb(o) > r$ then
                $C \leftarrow C \setminus \{o\}$
            end
        end
    end
end
```

Algorithm 4.3: kNN query with AESA access method

The algorithm operates with a candidate set C. At the beginning, this candidate set comprises the entire database and one candidate c is taken from this set in every iteration. There are many ways, how to select c from C. In the first iteration, the candidate is usually selected randomly. In the following iteration, some kind of heuristics may be applied, like selecting an object o with the smallest lower bound estimate $lb(o)$. Objects with smaller lower bounds are more likely to have smaller real distances, thus they are more likely to be included into the result set R and decrease the filtering range.

The lower bound estimates $lb(o)$ are kept in an array, which is initialized to zeros before the algorithm starts. These values are updated every time new query-to-object distance is computed. After this update, the candidate set is pruned using triangular inequality and current filtering range obtained from R.

Note that the range query algorithm would require only a minimal modifications as it uses constant filtering range r.

Even though this method is optimal in the number of computed distances, the lower bound updates and candidate set filtering can be quite expensive. Furthermore, the database indexing have $\Theta(|D|^2)$ both time and space complexity. With these limitations, the AESA method can be successfully used only in models with very expensive distance function and for rather small databases.

Linear AESA

A modification called *Linear AESA* (LAESA), presented by Micó et al. [94] tries to reduce the time and space complexity of the indexation by restricting the size of the pivot set. Only limited number of pivots P are selected from the database and the distances are precomputed between each pivot and every database object ($\forall p \in P, o \in D : d_P(p, o) \leftarrow d(p, o)$). The distances are stored in array of $|P| \cdot |D|$ items, which is called the *pivot table*. Let us emphasize that $P \subset D$, therefore distances between every two pivots are precomputed as well.

Standard LAESA algorithm is derived from the original AESA (Algorithm 4.3). The sole difference is, that the update of $lb(o)$ values is not performed in every iteration, but only if $c \in P$. It is also reasonable to alter the premise upon which the candidates are selected from the set C, so the pivots are selected in precedence to regular objects.

In case the number of pivots is rather small, we can use a 2-phase LAESA algorithm instead. In the first phase, we compute distances between query object and all pivots. These distances are used to create initial top-k result (or at least part of it, if $k > |P|$) and for pivot prefiltering in the second phase. The second phase traverses remaining objects in the database and computes the result. Each object goes through two step filtering. The prefiltering uses triangular inequality to quickly rule out too distant objects, while the regular filtering computes the distance function to determine, whether the object is included into the result. The Algorithm 4.4 formalizes the 2-phase approach. In the following, we denote the 2-phase LAESA access method as *pivot table prefiltering* since it is basically a simple traversal of the pivot table.

The pivot table prefiltering is clearly suboptimal in the number of computed distances as we do not use any type of prefiltering in the first phase. On the other hand, the 2-phase approach have some benefits. First of all, we do not require additional space for lower bound estimates, as these estimates are computed on the fly. Furthermore, the pivot table is traversed sequentially and only once, which leads to better utilization of the memory and processor caches. Finally, the pivot table prefiltering can be more easily parallelized[1] than original AESA and LAESA methods.

[1] Actually, this holds for range queries. The kNN queries have inherent problems with parallelism. We address this issue more thoroughly in Section 4.6.

```
Data: database $D$, set of pivots $P$ (precomputed distanced $d_P$), query $q$, $k$
Result: list $R$ of the $k$ closest objects (and their distances)
$R \leftarrow \emptyset$
// 1st phase: compute all query-pivot distances
foreach $p \in P$ do
    compute $d(q, p)$
    add $p$ to $R$ and remove the most distant object from $R$ if $|R| > k$
end
// 2nd phase: use pivots in prefiltering
foreach $c \in D \setminus P$ do
    if $|R| < k \vee lb_P(q, c) \leq \max\{d(q, o)|o \in R\}$ then    // prefiltering
        if $d(q, c) < \max\{d(q, o)|o \in R\}$ then   // $d$ computed, filtering
            add $c$ to $R$ and remove the most distant object from $R$ if $|R| > k$
        end
    end
end
```

Algorithm 4.4: kNN query with 2-phase LAESA access method (a.k.a., the pivot table prefiltering)

Performance Concerns of Pivot Table Prefiltering

As mentioned earlier, the pivot table prefiltering assumes, that the number of pivots is reasonably small. In our case, we are using only up to hundreds of pivots. The number of pivots is quite important. More pivots produce better lower bound estimates for the prefiltering, but on the other hand, the computation of a lower bound $lb(q, o)$ takes $\mathcal{O}(|P|)$ time.

An optimization technique called *early termination* can be employed to improve performance of the prefiltering step. The predicate $lb_P(q, c) \leq r$, where r denotes filtering range ($r = \max\{d(q, o)|o \in R\}$ in case of kNN query) can be replaced with $\exists p \in P : lb_p(q, c) \leq r$. Obviously, $\exists p \in P : lb_p(q, c) \leq r \Rightarrow lb_P(q, c) \leq r$, thus the semantics of the prefiltering is not altered. However, if we look for any p, that satisfies $lb_p(q, c) \leq r$, we can terminate as soon as we find one. This does not help in case no such p exists, but if it does, about half of the computation time can be saved in average.

4.2.3 List of Clusters

Another method of indexing is data partitioning or clustering. The general idea is to divide the data into fragments with common properties. In case of metric spaces, objects are divided according to their relative distances. The most direct approach is to partition data into *clusters* and organize these clusters as a simple *list* (LC index) [95].

A cluster i is defined as an ordered triplet (c_i, r_i, O_i), where $c_i \in D$ is the *center* of the cluster, $r_i \in \mathbb{R}^+$ its *radius*, and $O_i \subset D$ is a set of objects that belong to the cluster (sometimes called a *bucket*). All objects in the cluster falls within a metric ball represented by the center and the radius of the cluster (i.e., $\forall o \in O_i : d(c_i, o) \leq r_i$). The system of subsets O_i is in fact a decomposition of

D, thus $\bigcup O_i = D$ and $\forall i \neq j : O_i \cap O_j = \emptyset$. An example of the list of clusters is depicted in Figure 4.5.

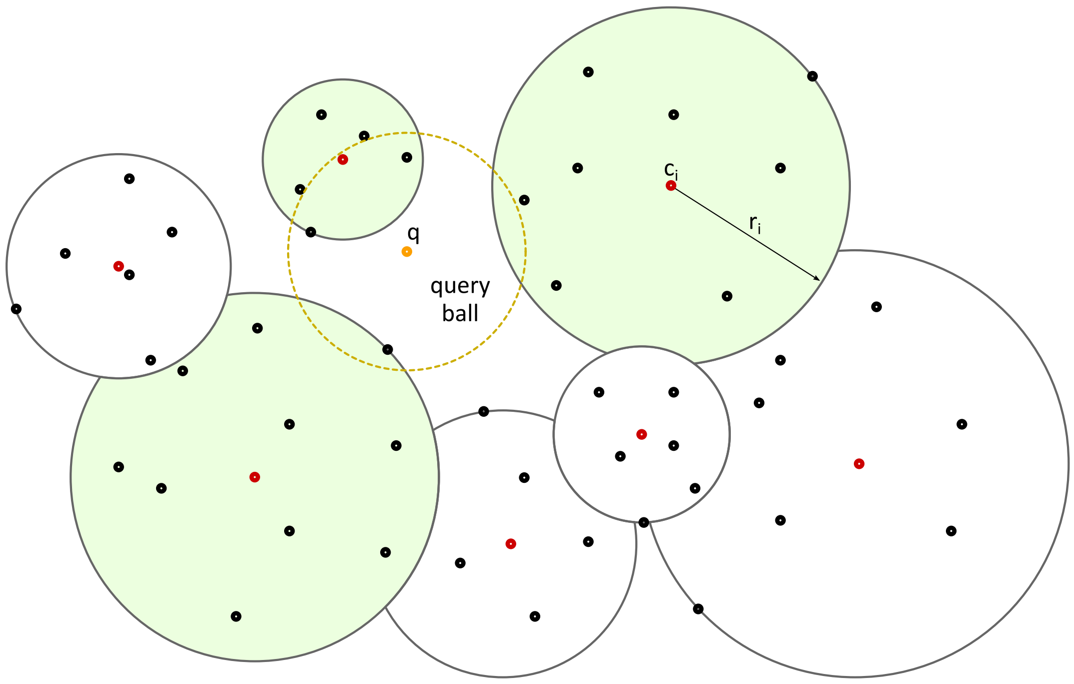


Figure 4.5: List of clusters (LC index)

There are various techniques and algorithms, how the clusters can be created. Usually, the list is constructed in a way, that all the clusters (except for the last one) have the same amount of elements. This corresponds to the block-oriented data organization of the persistent memory, where each cluster must be stored in a block of fixed size. However, there are also other approaches, such as constructing clusters with fixed radius. These algorithms are covered in the work of Chávez et al. [95].

Query Evaluation

Let us have a list of clusters index on top of our similarity model. The triangular inequality can be used to rule out whole clusters in case the ball of the cluster and the filtering ball of the query do not intersect. Formally, if $d(q, c_i) > r_i + r$, where r is the current filtering range of the query q, the entire cluster i (i.e., all objects in O_i) can be dismissed. The code is presented in Algorithm 4.5.

If the objects of one cluster are indeed stored together in the persistent memory, the list of clusters effectively reduce also the number of blocks read from this memory. On the other hand, the clustering is much more complicated and time consuming than simple computation of pivot table.

```
Data: database $D$, list $L$ of clusters $(c_i, r_i, O_i)$, query $q$, $k$
Result: list $R$ of the $k$ closest objects (and their distances)
$R \leftarrow \emptyset$
foreach $(c_i, r_i, O_i) \in L$ do
    if $|R| < k \vee d(q, c_i) \leq r_i + \max\{d(q, o)|o \in R\}$ then // prefiltering
        foreach $x \in O_i$ do
            if $d(q, x) < \max\{d(q, o)|o \in R\}$ then // filtering
                add $x$ to $R$, remove the most distant obj. from $R$ if $|R| > k$
            end
        end
    end
end
```

Algorithm 4.5: kNN query with list-of-clusters index

4.2.4 M-tree and PM-tree

The clustering approach can be extended by organizing the clusters into tree structures. Tree structures are adopted by many database systems, for instance B-trees [96] are used in relational databases or R-trees [97] are used for geometrical data. Similar decomposition can be employed in metric spaces, although there are some limitations. We present the *metric tree* and its improvement that combines tree structure with pivot table.

M-Tree

The metric tree (M-tree) was first introduced by Ciaccia [98]. It is quite similar to R-tree [97], but it uses balls instead of bounding rectangles to partition the metric space. Leaf nodes of the tree contain *ground entries* of the indexed data objects while inner nodes contain *routing entries.* A ground entry is a pair $\mathcal{G}_o = (o, d(o, Par(o)))$, that contains the object descriptor o and the computed distance between o and its parent object $Par(o)$ of the tree hierarchy. The routing entry is a tuple $\mathcal{R}_o = (o, d(o, Par(o)), Child(o), r_o)$. Again, the o stands for object descriptor and $d(o, Par(o))$ is the distance to the parent object. The $Child(o)$ is the reference to the child node in the tree and r_o is the covering radius.

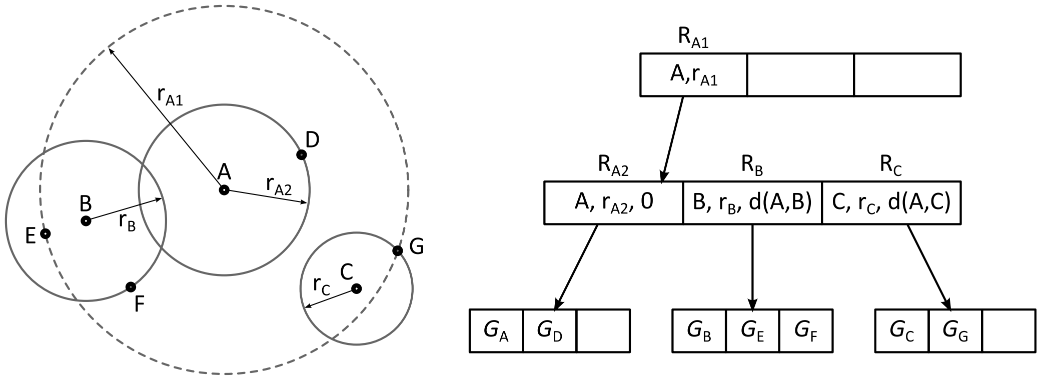


Figure 4.6: M-tree indexing example

All objects within a subtree represented by a routing object $\mathcal{R}_o$ must fall into the ball inducted by object o and radius r_o. Formally, $\forall \mathcal{R}_{o_i} \in \mathit{Tree}(\mathcal{R}_o) : d(o, o_i) \leq r_o$ and analogically $\forall \mathcal{G}_{o_i} \in \mathit{Tree}(\mathcal{R}_o) : d(o, o_i) \leq r_o$. This condition is quite weak as it allows to construct many different trees on the same data. An example of object set and one of the possible M-trees is depicted in Figure 4.6.

A technique similar to the one used by the LC index is used for prefiltering. Internal nodes of the tree are examined as if they were list of clusters. A distance between query and object o is computed for each record $\mathcal{R}_o$ of the internal node. If $d(q, o) \leq r_o + r$, where r_o is the covering radius of $\mathcal{R}_o$ and r is the filtering range of the query, a $\mathit{Child}(o)$ node is examined recursively. Otherwise, the entire subtree of $\mathcal{R}_o$ can be dismissed. When the recursion reaches leaf nodes, a distance is computed for each object o of the ground records $\mathcal{G}_o$ to determine, whether the object is included into the result set. A recursive implementation of the search process is formalized in Algorithm 4.6.

```
**Data**: database $D$, M-tree $T$, query $q$, $k$
**Result**: list $R$ of the $k$ closest objects (and their distances)
**function** *SearchNode(node $\mathcal{N}$, query $q$, $k$) $\rightarrow$ list of objects* :
    $R \leftarrow \emptyset$
    **if** $\mathcal{N}$ *is leaf* **then**
        **foreach** $\mathcal{G}_o \in \mathcal{N}$ **do**
            **if** $|R| < k \vee d(q, o) < \max\{d(q, x) | x \in R\}$ **then**    // filtering
                add $o$ to $R$, remove the most distant obj. from $R$ if $|R| > k$
            **end**
        **end**
    **else**
        $r \leftarrow \max\{d(q, x) | x \in R\}$
        **foreach** $\mathcal{R}_o \in \mathcal{N}$ **do**
            **if** $|R| < k \vee d(q, o) \leq r_o + r$ **then**                  // prefiltering
                $R \leftarrow R \cup$ SearchNode($\mathit{Child}(o), q, k$)
            **end**
        **end**
    **end**
    **return** $R$
**end**
// Initial search call
$R \leftarrow$ SearchNode($\mathit{Root}(T), q, k$)
```

Algorithm 4.6: kNN query with M-tree index

M-tree also introduces algorithms for updating operations. However, these operations are little concern to us as we focus on the acceleration of the searching problem. These methods can be found in related literature [98].

Pivoting M-tree

One of the major problems of the M-tree is that the bounding balls in metric space are not shaped in any way, so they cannot reflect the data distributions. The bounding ball usually covers only a few objects and a lot of *empty space*, which is sometimes referred as *dead space*. One of possible improvements is to crossbreed M-tree with pivot-based methods (like earlier introduced AESA and LAESA) to create *Pivoting M-tree* (*PM-tree*) [99].

The PM-tree is an extension of M-tree, so we describe only the differences. First of all, a set of pivots $P \subset D$ must be selected from database. The set is fixed for the lifetime of the index. The routing entries $\mathcal{R}_o$ in the internal nodes are extended by attribute HR, which is an array of *hyper-rings*. The ground entries $\mathcal{G}_o$ in the leaf nodes are extended by attribute PD, which is a list that stores distances to pivots.

Elements of hyper-rings $HR[i]$ holds the smallest covering distances between pivot $p_i \in P$ and all objects covered by corresponding routing entry $\mathcal{R}_o$. The $HR[i]$ record is an interval $\langle HR[i].min, HR[i].max\rangle$ in which the min and max values are in fact the minimum and the maximum of the set $\{d(o, p_i) | o \in O_{\mathcal{R}_o}\}$, where $O_{\mathcal{R}_o}$ stands for set of all objects in the subtree covered by routing entry $\mathcal{R}_o$.

The PD attribute of the grounding entry $\mathcal{G}_o$ is an array that holds precomputed distanced $PD[i] = d(o, p_i)$ between each pivot $p_i \in P$ and the object o. We can perceive this list as one row of the pivot table, so the pivot table is in fact scattered over the leaf nodes. Let us emphasize that both HR and PD arrays have $|P|$ items, thus the number of pivots must be reasonably small in order to keep the size of the PM-tree within feasible limits.

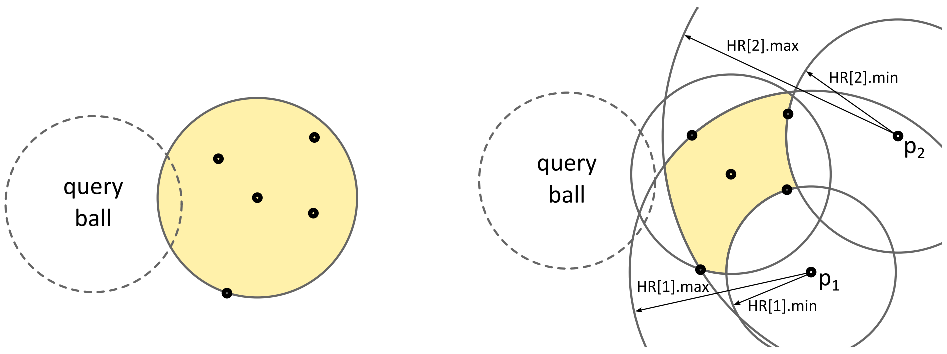


Figure 4.7: How pivots improve M-tree partitioning

An example that demonstrates, how the pivots improve partitioning of the metric space is depicted in Figure 4.7. Each hyper-ring acts as a new boundary that cuts of some dead space and the covering space of the node is in fact an intersection of the original bounding ball and all the hyper-rings.

The query algorithm is quite similar to the Algorithm 4.6 for M-trees. Before the algorithm is executed the distances $d(q, p_i)$ between the query q and all the pivots $p_i \in P$. In the algorithm itself, the prefiltering condition, which is evaluated for each routing entry, is enriched by the following formula:

$$\bigwedge_{p_i \in P} d(q, p_i) + r \geq HR[i].min \wedge d(q, p_i) - r \leq HR[i].max$$

It tests whether the query ball (defined by the query q and filtering range r) intersects with the area determined by the hyper-rings. If the formula evaluates as false, the subtree of the routing entry does not have an intersection with the query ball, thus it can be skipped. Furthermore, a prefiltering test is added for the ground entry case:

$$\bigwedge_{p_i \in P} |d(q, p_i) - PD[i]| \leq r$$

This is in fact the same test, which is performed in the prefiltering step of the 2-phase LAESA method (the pivot table filtering). If the second formula resolves as false, the distance between query and corresponding object in the ground entry needs not to be computed.

4.2.5 M-Index

A completely different approach takes the *Metric Index* (M-Index) [100]. The concept of M-Index is based on iDistance [101], which is also a similarity search indexing method, but it is designed for vector spaces. The general idea is to create a linear ranking for the objects that provides each object with numeric key from $\mathbb{R}^+$. This key is then used to store objects into a well-established data structures, a B$^+$-tree [96] for instance.

The index expects the database to be partitioned by a set of pivots $P \subset D$, which is selected from the database when its being indexed. A Voronoi-like partitioning is used to divide the database into $|P|$ clusters. Each object is assigned to its closest pivot and objects assigned to one pivot p_i create a cluster C_i. Given a constant c great enough to separate distinct clusters (i.e., $c > \max\{d(p_i, o) | \forall p_i \in P, \forall o \in C_i\}$), an *iDistance ranking* for object $o \in C_i$ can be defined as

$$iDist(o) = d(p_i, o) + ic.$$

Each cluster C_i ($i = 0, 1, \ldots$) has its own interval $\langle ic, (i + 1)c \rangle$, thus the objects from the same clusters are mapped close together and their distance to the corresponding pivot is reflected in the iDistance value. If the metric d is normalized ($d : D \times D \to \langle 0, 1)$), we omit the multiplicative constant ($c = 1$) and each cluster is neatly mapped between two natural numbers. The principle of linear ranking is depicted in Figure 4.8.

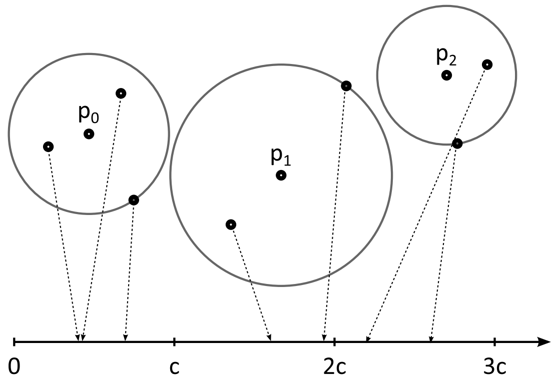


Figure 4.8: Principle of iDistance ranking employed by M-Index

Thanks to the Voronoi partitioning and the *double-pivot distance constraint* [90], cluster C_i can be skipped if $d(p_i, q) - d(p_j, q) > 2r$ for any $p_j \in P$. If the cluster C_i is not ruled out by the prefiltering, its objects can be easily retrieved from the B$^+$-tree by applying simple interval query for $\langle i \cdot c, (i+1) \cdot c)$ keys. The B$^+$-tree respects the spatial locality of the keys, thus the results are most likely compacted in only a few tree nodes.

The idea presented above reflects a *one level* M-Index. In case the data are really large, the index can be extended to a tree hierarchy of the Voronoi partitioning. The number of levels need not to be static, hence a dynamic partitioning can be employed to better reflect the structure of the data. Finally, the M-Index may be used also for an approximate query search in addition to traditional range and kNN queries. As these improvements are quite complex, they are well beyond the general introduction, thus we did not present them here. All the details, including formalization of algorithms, can be found in the related literature [100].

4.2.6 Ptolemaic Spaces

The metric access methods described earlier are based on the axiom of triangular inequality. If the distance function conforms to stronger axioms than metric ones, these axioms may be used to design more elaborate and more effective indexing methods. In this section, we introduce the *Ptolemaic indexing* [102, 103] based on *Ptolemaic metric.*

The indexing is based on the *Ptolemy's inequality.* It states that for any quadrilateral, the pairwise products of opposing sides sum to more than the product of the diagonals. In other words, for any four points $x, y, u, v \in \mathbb{U}$ the inequality formulates as the following:

$$d(x,v) \cdot d(y,u) \leq d(x,y) \cdot d(u,v) + d(x,u) \cdot d(y,v)$$

A distance function that satisfies identity, nonnegativity, symmetry, and Ptolemy's inequality is a *Ptolemaic distance.* If a Ptolemaic distance also conforms to triangular inequality axiom, it is a *Ptolemaic metric.* The SQFD function satisfies the Ptolemy's inequality [103], hence it is a Ptolemaic metric.

We can use the Ptolemy's inequality in a similar way we used the triangular inequality. Let us have a set of pivots P and precomputed pivot table d_P. A lower bound estimate can be computed for each pair of pivots from P. For given query q, object o, and pivots $p, s \in P$, the lower bound is defined as:

$$lb_{p,s}(q,o) = \frac{|d(q,p) \cdot d_P(o,s) - d(q,s) \cdot d_P(o,p)|}{d_P(p,s)}$$

We expect that $p \neq s$, thus $d_P(p,s) > 0$ since the distance function has the identity and nonnegativity properties. The value of $lb_{p,p}(q,o)$ (i.e., the single-pivot Ptolemaic estimate) can be additionally defined as zero for the sake of completeness.

The prefiltering step of the AESA or the pivot table prefiltering algorithm needs only to update the condition to compute a lower bound estimate for each pair of pivots rather than for each pivot. Other metric access methods would require more elaborate modifications, which we do not cover here as we are focusing mainly on the pivot table prefiltering method.

The complexity of the prefiltering step increases from $\mathcal{O}(|P|)$ to $\mathcal{O}(|P|^2)$ with the Ptolemaic inequality. On the other hand, Ptolemaic prefiltering requires smaller amount of pivots to achieve the same effectiveness as the triangular prefiltering [103].

We have introduced two types of lower bounds, both denoted *lb*. If there is need to distinguish between them, we use $lb^{\triangle}_{p}(q,o)$ for triangular and $lb^{Pt}_{p,s}(q,o)$ for Ptolemaic lower bound. In addition, we define $lb^{\oplus}_{P}(q,o)$ as the *combined* lower bound. The combined lower bound uses the better (i.e., higher) estimate of the other two (triangular and Ptolemaic) lower bounds.

4.3 Related Work

In this section, we revise some of the work related to parallelism in similarity search. We focus especially on multimedia similarity search and GPGPU parallel architectures.

Parallel Similarity Search

Individual parallel algorithms, which can be used to evaluate distance functions, were established long time ago. For instance, a parallel algorithm for computing the edit distance was introduced in 1988 by Mathies [104].

The first attempts to utilize parallel architectures for similarity search emerged in the field of bio-computations. This field struggles with many problems that employ similarity search techniques, such as searching the protein databases or DNA sequences. Such comparisons are very expensive and they can really benefit from the parallel execution.

Galper et al. [105] addressed the problem of alignment which is usually solved by dynamic programming. They have pointed out that most of the parallel algorithms for this problem are designed for theoretical architectures (such as PRAM [106]), so they proposed several practical approaches for a multiprocessor shared-memory system.

Gish et al. [107] explored the BLASTX technique which identifies protein coding regions in nucleotide sequences. As this method is quite time consuming, they proposed a parallel implementation of BLAST for multiprocessor systems.

One of the most recent endeavours of Galgonek et at. [108] was the parallel implementation of the SProt measure, which is used for structural comparison of two proteins. Their implementation achieved almost linear speedup on 4-node NUMA server with multi-core CPUs using Intel Threading Building Blocks library. They have also parallelized the TM-score algorithm, which is used in many other similarity search approaches.

Multimedia Similarity Search

The situation is a little different in case of parallel multimedia similarity search. The sequential scan is an embarrassingly parallel problem and the internal parallelization of distance computations is usually less efficient on multi-core CPUs. Furthermore, there has been no attempts to employ other architectures (such as GPGPU) for parallel distance computations to our best knowledge.

When the metric indexing is employed, it gets slightly more interesting. The pivot table prefiltering is still considered to be almost ideal data parallel problem.

Even though the kNN query suffers from the filtering range update dependency (which we address in Section 4.6), it performs rather well on multi-core CPUs. Other parallel and distributed metric index methods are summarized in the book of Zezula [90].

The most interesting of these methods is the parallel implementation of M-tree [109]. It is based on optimal declustering, which considers the object proximity to balance the workload and also reduce the disk I/O. However, the algorithm does not address the problem of shared priority queue, which is used to hold the intermediate top-k result of a kNN query.

When the scope of the database exceeds the capabilities of a single server or tightly coupled cluster, a distributed approach is required. The first distributed index for metric spaces was presented by Batko et al. [110]. It is based on the Generalized Hyperplane Tree (GHT*) where the nodes of the tree are distributed across multiple computers. The data structure is designed so that many operations (including node split) can be performed locally, thus efficiently.

Another example of distributed solution is the Metric Chord (M-Chord) [111] data structure. The general idea is to index objects in metric space by iDistance [101], like in the case of M-Index [100]. Then a Chord [112] distributed data structure is used to store the objects.

Nearest Neighbours Query Problem

The most fundamentally studied problems that extends beyond the realms of similarity search is the k nearest neighbours, also known as the top-k problem. One of the first parallel approaches to nearest neighbour search [113] was proposed by a research team from Munich university in 1997. In the work, they assumed that the object descriptors are mapped to high dimensionality spaces and compared by rather cheap distance functions. They proposed to cluster the feature space, so the clusters may be searched concurrently.

To the best of our knowledge, the first implementation of kNN query on GPUs was presented by Bustos et al. [114] in 2006. Their implementation was restricted to compute k nearest neighbours using Manhattan distance in $\mathbb{R}^d$ spaces, where d varied up to 256. The work proposed a GPU-specific data representation, which allowed better utilization of the texture cache (local memory) of the SMPs.

A similar approach was taken by Garcia [115] in 2008. Their experiments tested Euclidean and Manhattan distances in $\mathbb{R}^d$ spaces for dimensions between 8 to 96. The brute force implementation outperformed not only naïve serial algorithm, but also the version which used kd-tree to index the space [116].

The most recent work on the same topic was done by Barrientos et al. [117]. They improved the performance by replacing parallel sorting with standard 2-regular heap that holds the intermediate top-k result. The GPU parallel implementation was based on heap reduction. The heap is kept replicated, so each thread in a warp has its own copy and the data from multiple heaps are then combined by a parallel reduction algorithm.

4.4 Accelerating SQFD by GPUs

In this section, we describe our contributions to the field. We have implemented a similarity search framework that allows us to compute the Signature Quadratic Form Distances of modern GPU cards.

4.4.1 Problem Analysis

Before we describe of our GPU solution, let us address several issues regarding the integration, hardware properties, and data properties.

Implementation Requirements

An image database system that employs similarity search consists of many parts, such as persistent storage, indexing modules, etc. Our first concern regards integration with these parts. The GPU implementation must be completely independent on the other parts of the system. On the other hand, the GPU requires the signatures to be stored in specific format in order to achieve the best performance possible. This modification can be easily propagated to the system as it does not affect anything else.

The GPU SQFD version is expected to be used with different types of queries, especially the range queries and the kNN queries in combination with some kind of metric access methods which perform prefiltering of the database. It can also be used for indexing methods, which require to precompute some of the distances, like pivot table construction or object clustering. In all cases, many distances are expected to be computed to evaluate one query or to build an index. We cannot hope to fit object signatures of the entire database into memory of the GPU. On the other hand, issuing a task for a GPU is bound with nontrivial overhead, thus computing each distance in a separate GPU task would be highly inefficient. The only reasonable compromise in this case is to bundle signatures into blocks and let GPU compute multiple distances in each task.

There are several possibilities, how to implement block-wise distance computations. The most direct approach is to send $2N$ paired signatures and compute N distances. In case there is only one query, half of these signatures will be a copy of the query signature. The possible utilization scenarios need to be explored in order to select the best model. There are three imaginable situations:

- single-query evaluation,
- multi-query evaluation,
- full object-to-object distance matrix enumeration.

Single-query and multi-query evaluations are quite common. Multi-query model can increase the processing throughput and it reduces the data transfers between GPU and the host system as each object signature transferred to GPU is used multiple times for multiple queries. Multi-query model is also used when pivot table is constructed as we compute distances between database objects and pivots. The situation, when complete distance matrix is computed is slightly more rare, but it can occur during database clustering or similarity model training for instance. It is also used, when the results of a query are being clustered for a better visualization.

Hardware Considerations

From the hardware point of view, most of the issues have been already summarized in Chapter 2. Only a few things remain to emphasize. First of all, the host system may be equipped with more than one GPU. The GPUs present may not even be of the same type. The implementation should utilize all of the present GPUs to the best of their abilities. On the other hand, no GPU may be present in the system. In such case, the fallback to original CPU implementation has to be possible.

Furthermore, the distance computations should leave the rest of the database system free to perform other tasks, such as filtering the results or prefiltering the candidates. Therefore, the block-wise distance computation has to be implemented as an asynchronous operation from the programmers point of view.

Data Overview

The distance function is computed between two feature signatures. The feature signatures were described at the end of Section 4.1.2. We need to be aware of common signature sizes of the real-world data in order to design an optimal algorithm. Figure 4.9 presents histograms of signature sizes gathered from our image datasets. These histograms suggest that most of the signatures have between 50 and 100 centroids and the largest signature encountered had 230 centroids.

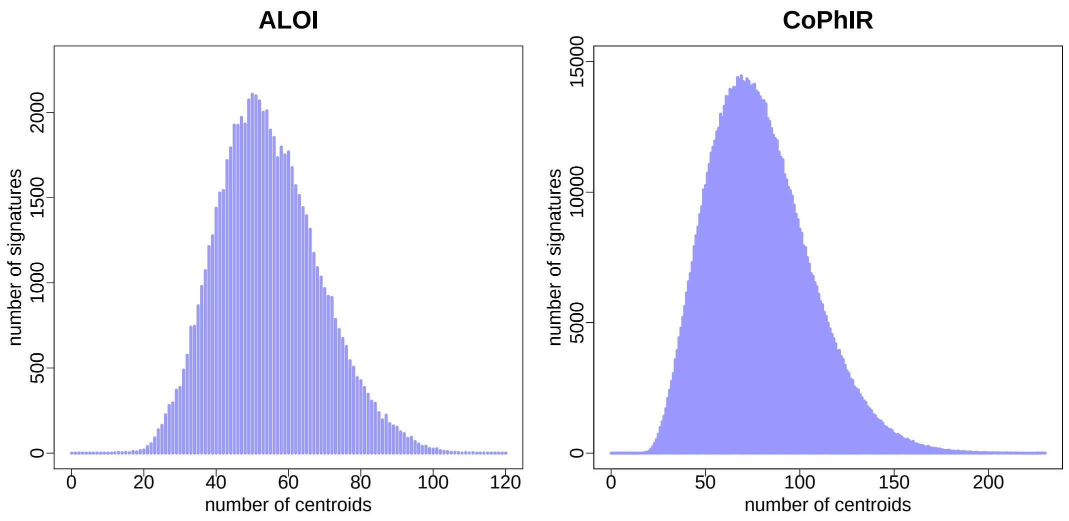


Figure 4.9: Histograms of signature sizes of two real-world datasets

We use 7 dimensional feature space, where each feature is represented by one float number. The weights correspond to the number of features in their respective clusters, so they could be stored as integers; however, the SQFD requires the weights to be normalized ($\sum w_i = 1$), so we store them also as float numbers. Empirical results indicate that single precision floats (32 bits) provide sufficient accuracy for the computations, hence one centroid of the signature takes $(7 + 1) \times 4\text{B} = 32\text{B}$. As most signatures have 50 to 100 signatures, they take $1,600$ to $3,200$ bytes of memory space.

Both centroid values and weights are accessed multiple times during the computation. Therefore, the signatures should be cached in the local memory of the SMP. The Fermi architecture offers us 48 kB of space, which can be used to store two signatures of $(49,152/2)/32 = 768$ centroids. According to signature size histograms, this amount is more than enough for our signatures.

Conclusions

As mentioned before, the distances should be computed in a block-wise manner. The problem we are solving fits the description of an *iterative task* described in Section 3.4.2. Therefore, we can use our OpenCL wrapper with CPU feeding threads, which was described in Chapter 3. This framework helps us to meet our implementation criteria and deal with the problem of balancing the workload amongst multiple GPUs.

Concerning the workload inside each GPU, a slightly more elaborated plan will be required. Despite the fact there might be enough signatures in the block to occupy all the GPU cores, assigning one distance to each thread is not optimal. The signatures need to be cached in the local memory of the SMP (shared by all threads in the group) which can accomodate only a few signatures. Furthermore, signatures have different sizes and, so the workload would be very imbalanced amongst the threads. All threads in a warp would have to wait for the thread with the larges signature as the threads in a warp execute their code in lockstep.

For these reasons, we have decided to use two levels of parallelism. The distances of signatures in one block are computed concurrently by thread groups and threads in a group compute one distance cooperatively. The principles of signature dispatching and parallel computation of distances are described in Section 4.4.2. The details concerning how one SQFD distance is computed in parallel by threads employing SIMT model are described in Section 4.4.3.

4.4.2 Similarity Search in Parallel Environment

The similarity search engine was implemented on the top of OpenCL wrapper combined with feeding threads. The architecture of the system is depicted in Figure 4.10. The core part comprises of prefiltering and filtering. The prefiltering is responsible for generating candidates, which are formed to blocks and dispatched to the input queue of the GPU feeding thread pool. Computed distances are recovered from the output queue of the feeding pool and filtered. The filtering process maintains a set of partial results until all distances for all candidates are computed.

In this description, we have generalized the term of prefiltering. Usually, the prefiltering refers to an application of some metric access method (e.g., the pivot table prefiltering). If no special access method is used and a simple sequential scan is performed, the prefiltering only sequentially gathers signatures and pack them into blocks.

The performance of the system can be affected by the block size and how many blocks are dispatched concurrently. The main thread operates in a loop

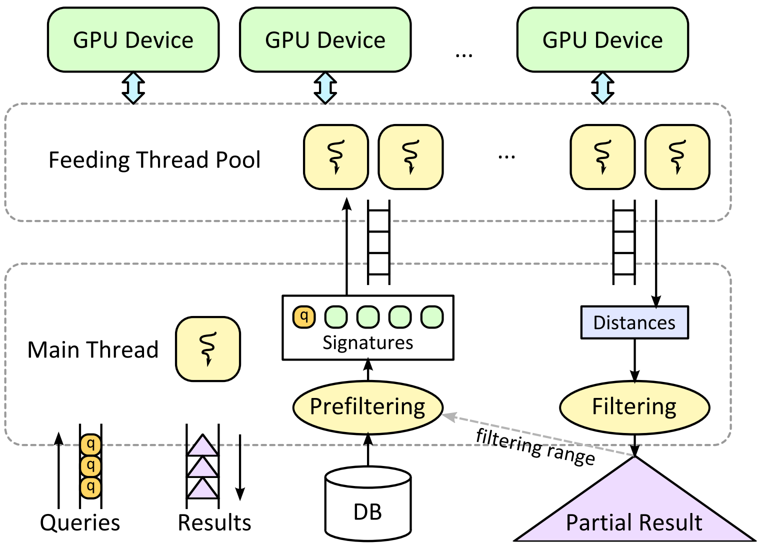


Figure 4.10: Similarity search framework with GPU acceleration

over entire database performing prefiltering. If a block of candidates of sufficient size is ready, it can be dispatched to the feeding threads. If a block of distances is available, the prefiltering may be interrupted to filter the distances and update the partial result set. The main thread can also choose to wait for next distance block in case there are enough signature blocks currently dispatched. The exact strategy is also dependent on the type of the query being solved.

In case of the range query, there is no particular reason to wait for the distances, except the memory limitations imposed on the feeding thread queues perhaps. However, the kNN query is a much more complicated matter. The filtering range is updated every time a block of distances arrives and the current top-k result is updated. Unfortunately, the filtering range is also required by the prefiltering step. In fact, it would be best to compute distances sequentially as each distance can contribute to improving the filtering range. We address this issue more thoroughly in Section 4.6.

Block Types

To cover all possible scenarios of usage, we have designed two types of blocks:

- a multi-query distance block
- and distance matrix block.

The former block type contains N_q query signatures and N_o object signatures. The result comprises all distances between each query-object pair (i.e., $N_q \cdot N_o$ values). Usually, the query signature block has only one query. The latter block type contains N signatures and the entire distance matrix is computed. To save space, we compute only one half of this matrix as the distance function is symmetric, so the result consists of $N(N-1)/2$ distances.

As mentioned before, each distance is computed by one thread group. The multi-query case dispatches $N_q \cdot N_o$ groups and i-th group computes distance between query $q_{\lfloor i/N_o \rfloor}$ and object $o_{(i \bmod N_o)}$. The distance matrix case requires slightly more elaborated mapping as only half of the matrix is computed. The mapping is depicted in Figure 4.11.

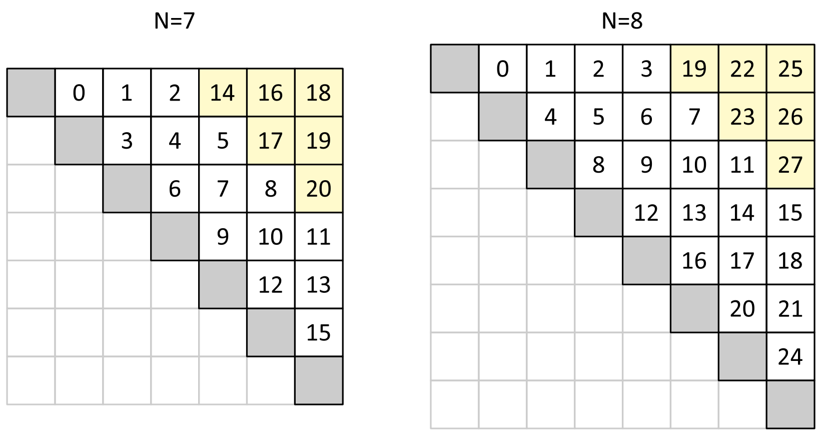


Figure 4.11: Examples of group id mappings to the distance matrix

This mapping was chosen, so that each thread can compute the indices of the objects o_i, o_j from the group index G_{ID} with only a few integer operations. The Algorithm 4.7 presents the formulas used for computing object indices from group identifier.

```
Data: group index $G_{ID}$, number of objects $N$
Result: object indices $i$ and $j$
$i \leftarrow \lfloor G_{ID} / \lfloor N/2 \rfloor \rfloor$
$j \leftarrow G_{ID} \mod \lfloor N/2 \rfloor + i + 1$
if $j \geq N$ then                    // out of matrix borders check
    $i \leftarrow i + 1 - (N \mod 2)$        // special correction for odd $N$
    $j \leftarrow j - N$
end
if $i \geq j$ then     // make sure we compute upper triangular matrix
    swap $i$ and $j$
end
```

Algorithm 4.7: Computing matrix position from group index

The results are compacted to an array of size $N(N-1)/2$ in a row-wise manner, so the i-th row has only $N - i - 1$ items. The index i_{res} to the array can be computed from indices i, j as

$$i_{res} = \frac{N(N-1) - (N-i)(N-i-1)}{2} + (j - i - 1).$$

Feeding The GPUs

The feeding threads prepare the data for the GPU, upload them to GPU buffers and execute the SQFD kernel on the GPU. When the kernel finishes, the distances are copied back to prepared buffer in the host memory. The problem is, that each memory transaction to the GPU is bound with serious overhead. The data must be prepared in one (or at least in very few) continuous block, otherwise the performance drops dramatically[2]. The feeding threads are responsible for copying signatures from database buffers into a temporary buffer and then issue a transfer to GPU.

Another problem is, that all the threads computing distances of one block of signatures are provided only a pointer to the beginning of the buffer with the signatures. Each thread group require two signatures from this block, but the signatures are not aligned in the buffer as they differ in sizes. In order to find these signatures quickly, an index is built by the feeding thread during the copying step. The index contains offsets of all signatures in the block and the size of the block. Hence, signature for object i starts at position $Idx[i]$ and its size is $Idx[i+1] - Idx[i]$. The index is transferred to the GPU along with the buffer containing feature signatures.

We have mentioned that the SQFD can be optimized by precomputing the self-similarity submatrices of all database objects and the query. As we show later in Section 4.4.3, the self-similarity value can be stored as single number. The feeding thread assembles the self-similarity values for all signatures in the block and prepares them into separate buffer, which is transferred to the GPU along with signatures and their index.

Signature Format

The final issue regards the feature signature representation. Let us revise, that each signature S^o is a list of centroids, where each centroid c_i^o is a point in $\mathbb{R}^7$ space and a weight $w_i^o \in \mathbb{R}^+$ is attached to each centroid. Each value is represented as 32 bit float number, which happens to be an ideal choice for the GPU, as it operates naturally with 32-bit values.

The data format does not concern the rest of the system, because the remaining parts regard the signatures as blocks of binary data. The format has also little consequence for the CPU implementation as the signature usually fits the L1 cache, thus any compact representation works fine here. On the GPU, the signatures are cached manually in the local memory of the SMP. The SMP is divided into banks, so that two consecutive 32-bit words are in two following banks. The peak performance is achieved when different threads access data in different banks.

It is a good practice to organize this type of data as an array of structures, where each structure represent one centroid (7 coordinates) and its weight. The signature values are linearized in memory as $(c_1[1], c_1[2], \ldots, c_1[7], w_1, c_2[1], \ldots)$. Let us assume that all the threads read different centroid at once. In the first step, each thread loads $c_i[1]$ (where i is different for each thread). As the first value of each centroid is aligned to 8 (addressing in multiples of 32-bit values), threads on cores $0, 4, 8, 12, \ldots, 28$ will collide on bank #0, threads on cores $4k+1$ collide on bank #8 and so on.

To avoid this problem, we organize the data in column-wise manner (in a structure of arrays) as suggested in Section 2.2.4. Hence, there will be a block of $c_i[1]$ values ($i = 1, \ldots, |S^o|$) followed by $c_i[2]$ values and so on, enclosed by the

[2]Actually, several different approaches were tested, such as enqueueing transfers into out-of-order command queue or using memory mapping for GPU buffers. The presented approach was the fastest one.

block of w_i values. This way, the chance of collision is significantly reduced and if consecutive threads reads consecutive centroids, there are no collisions what so ever.

Another approach would be to separate only a vector of weights. The centroids will be stored as array of structures $(c_1[1], c_1[2], \ldots, c_1[7], c_2[1], \ldots)$ followed by an array of weights. As the centroids are aligned to 7 words and 32 is not divisible by 7, the probability of a collision is again significantly lower. However, our first approach works for a feature space of any dimension (not only for 7).

4.4.3 Parallel Implementation of SQFD

One SQFD distance is computed by one work group, which is mapped to one symmetric multiprocessor. The main restriction of this arrangement is that the algorithm needs to be rewritten to fully embrace the SIMT nature of the multiprocessor. This means that we need to avoid creating a lot of branches or while-loops and to balance the workload amongst the threads.

SQFD Revision

Let us briefly revise the definition of the SQFD (thoroughly described in Section 4.1.4) for the purpose of its optimization for GPUs. The distance is defined as:

$$d_{SQFD_{f_s}}(S^q, S^o) = \sqrt{(w_q| - w_o) \cdot A_{f_s} \cdot (w_q| - w_o)^T}.$$

The A_{f_s} is the similarity matrix defined by similarity function f_s. In our implementation, we have used $f_s(c_i, c_j) = e^{-\alpha \cdot L_2^2(c_i, c_j)}$, but it can be easily changed.

The *naïve approach* is to enumerate matrix A_{f_s} and compute the multiplication with the first vector $w = (w_q| - w_o)$. The product would have the same proportions as w and this intermediate result has to be stored in the memory. Then, it is multiplied with w^T and single scalar is yielded. Finally, the square root of this scalar is found, thus the distance is computed.

Exploring Implementation Alternatives

There are many approaches to parallel implementation. As it is hard to formally prove that an implementation is optimal or even better than another, we at least examine other feasible alternatives and explain, why they have been rejected. We start with a few observations, which are quite important for any implementation:

- The inputs can be cached into local memory and it is imperative they are cached. Therefore, we assume that every algorithm does so. On the other hand, it has to be noted that the signatures take significant portion of the local memory, so any algorithm have less room for intermediate results.
- The enumeration of f_s takes the most of the computational time, since it needs to be computed for each pair c_i, c_j.
- The A_{f_s} matrix needs to be enumerated on the fly. Local memory of the SMP cannot accomodate matrices larger than 110×110, which is not nearly enough according to our data analysis.

The naïve approach works very well on the CPUs as it uses optimal number of arithmetic operations. However, in massively parallel environment, several problems emerge. Elements of the similarity matrix can be enumerated independently, thus concurrently. The problem is, that in the first multiplication $v = (w_q | - w_o) \cdot A_{f_s}$, each column of the matrix contributes to only one item in the intermediate result v, so the access to values of v must be synchronized. The same problem rises in the second multiplication, where a sum of the products of corresponding items in the vectors v and w^T needs to be found.

To avoid synchronization implied by the naïve approach, we can try reordering the workload among the threads, so that each shared item is then managed by only one thread. In this case, we assign each element of the intermediate vector v to single thread in a round robin fashion. Hence, all the write operations are mutually exclusive. We denote this approach the *vector-parallel* algorithm as it parallelizes the work over the intermediate vector result.

The vector-parallel approach computes a vector v of the size $|S^q|+|S^o|$, where item v_i is computed as a sum of values from the i-th column of the A_{f_s} matrix multiplied by their corresponding weights from w. The values of A_{f_s} can be enumerated on the fly as they are immediately multiplied by weight and added to a partial sum. After that, each value v_i is multiplied by w_i and all the values are added up. The final addition can be performed either simply by only one thread, or by standard binary reduction tree technique.

Even thought the vector-parallel approach seems efficient as it does not require any synchronization (except for a simple barrier), it does not generate very balanced workload. If the size of the vector ($|S^q| + |S^o|$) is equal to $k \cdot T$, where $k \in \mathbb{N}$ and T is the number of threads in the work group, it will work perfectly. However, this is rarely the case as a fixed number of threads per work group must be used and the signature sizes vary significantly. The NVIDIA programming guide [34] suggest to use at least 4× as many threads as there are cores on SMP, so we should use at least $4 \times 32 = 128$ threads. Smaller signatures may cause that many threads will be idle. Furthermore, if the $|S^q| + |S^o|$ is not divisible by the warp size, there will be one warp in which some of the threads are working and some are just blocking available cores.

Any other column-wise or row-wise approach to dividing the A_{f_s} matrix will suffer the same problems as the vector-parallel algorithm. Better load balance must be achieved by computing all the elements of the similarity matrix concurrently. We can easily linearize the elements of the matrix and assign them to threads in round robin manner. This technique is also used in our approach, so we describe it later in more detail. The problem is, how to efficiently deal with the synchronization of the access to the intermediate result produced by the first multiplication. We have explored several possibilities.

The operation performed by the threads is in fact an atomic addition. Each thread computes a result of the f_s function, multiplies it with corresponding weight and adds it to a partial sum in the intermediate vector. Unfortunately, all the values are represented as float numbers and OpenCL does not provide atomic add for floats [38]. CUDA specifies atomic add on 32 bit floats; however, such solution would not be portable. We can emulate atomic add using atomic compare-and-exchange in a while loop, but such solution is quite slow.

We have tried to devise other methods how to mutually exclude the addition operations, but none of them was both thread-safe and efficient. A standard implementation technique would be to use *privatization.* The shared data are replicated, so each thread has its own copy. These copies are merged, when the main part of the algorithm finishes. Unfortunately, there is not enough room to fit so many copies of the vector in the local memory.

Our Approach

Our approach tries to take the best ideas from the presented alternatives and combine them together. The key difference is, that we resent the naïve approach and we do not construct the intermediate vector v of the first multiplication $w \cdot A_{f_s}$ in the local memory. For the purpose of our algorithm, we define a *weighted similarity matrix* $A^w_{f_s}$ as

$$A^w_{f_s} : a^w_{ij} = w_i \cdot a_{ij} \cdot w_j = w_i \cdot f_s(c_i, c_j) \cdot w_j,$$

assuming the $w = (w_q| - w_o)$ and $i, j \in \{1, \ldots, N\}$ if we denote N the rank of the matrix ($N = |S^q| + |S^o|$). According to laws of distributivity, the SQFD is then computed as

$$d_{SQFD_{f_s}}(S^q, S^o) = \sqrt{\sum_{a^w_{ij} \in A^w_{f_s}} a^w_{ij}} = \sqrt{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} w_i \cdot f_s(c_i, c_j) \cdot w_j}.$$

In other words, the items of the weighted similarity matrix are created by enumerating f_s function and multiplying the f_s values by their corresponding weights. The result is then produced as the sum of the values of the weighted matrix.

Proof. We need to prove, that this modification produces the same result as the originally defined SQFD formula. We start with the original vector-matrix multiplications beneath the square root. For better reading, the algebraic multiplication of two numbers is denoted $\cdot$ while vector and matrix multiplications are denoted $\times$.

$$\begin{aligned} w \times A_{f_s} \times w^T &= \sum_{j=0}^{N-1} (w \times A_{f_s})_j \cdot w_j = \sum_{j=0}^{N-1} (\sum_{i=0}^{N-1} w_i \cdot f_s(c_i, c_j)) \cdot w_j \\ &= \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} w_i \cdot f_s(c_i, c_j) \cdot w_j = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} w_i \cdot f_s(c_i, c_j) \cdot w_j \end{aligned}$$

The first two equivalences only apply rules of matrix multiplication. The third equivalence uses the law of distributivity, and the final one the fact, that addition on real numbers is an associative operation. □

Let us note that the presented proof is valid from a mathematical point of view. However, real numbers are approximated by their discrete representation with limited accuracy. It is known, that float numbers are not completely associative due to inherent rounding errors. We have confirmed empirically, that the results produced by the modified SQFD differs from the original results, but these differences are negligible and have no measurable effect on the similarity search precision.

The weighted similarity matrix approach produce the same result as the naïve approach, but it is clearly suboptimal in the number of arithmetic operations performed. As we used distributivity to multiply every element of the matrix with both weights, the number of float multiplications increased by $N^2 - N$. On the other hand, this increase is hardly measurable as each core has its own floating point unit and the computations in the f_s function surpass these additional multiplications by orders of magnitude.

Furthermore, our approach helps us better exploit the optimization presented at the end of Section 4.1.4. The weighted similarity matrix can be divided into four regions the same way the orignal similarity matrix was. Both self-similarity submatrices can be computed in advance, and since we are interested only in the sum of all the elements, we can save them as partial sums. Let us denote these partial sums σ_q and σ_o for the query q and the database object o respectively. The remaining two submatrices that represent the inter-similarity are in fact symmetric, since f_s is based on a metric distance. If we denote the partial sum of one inter-similarity submatrix $\sigma_{q,o}$, the distance can be computed as $d_{SQFD}(S^q, S^o) = \sqrt{\sigma_q + \sigma_o + 2\sigma_{q,o}}$.

Work Decomposition

It has been demonstrated, that we need to enumerate weighted matrix $A^w_{f_s}$ and sum up all the values in order to compute SQFD. All the elements of the matrix can be computed independently, so we need to focus only two things:

- how to decompose the matrix amongst the worker threads to achieve the best load balance
- and how to perform effective parallel summation.

The first problem is solved with straightforward linearization of the matrix and by assigning the items to threads in a round robin fashion. Each element a_{ij} of the matrix has linear index[3] $l = \mathit{width} \cdot (i-1) + (j-1)$, and each thread process elements with indices $kT + T_{ID}$, where $k = 0, 1, \ldots$, T is the number of threads in the group, and T_{ID} is the ID of a thread. This method is easily adopted for rectangular section of the matrix as well, so we can compute only the $\sigma_{q,o}$ of the inter-similarity submatrix. The partitioning principle is depicted in Figure 4.12.

The second problem is solved in two steps. We cannot have a single variable to hold the sum of the weighted matrix as it would become a serious bottleneck due to synchronization. Fortunately, the *privatization technique* can be used here, since only one variable needs to be replicated. A partial sum variable is allocated per thread in the local memory and each thread adds computed elements of the $A^w_{f_s}$ matrix to its private variable.

At the end, all the partial sums need to be added up into one final sum. This can be solved the same way as in the vector-parallel approach. A single thread

[3]Even though we normally use natural numbers for mathematical indices, the linear index is zero based for computational purposes. That is why we had to decrement i and j.

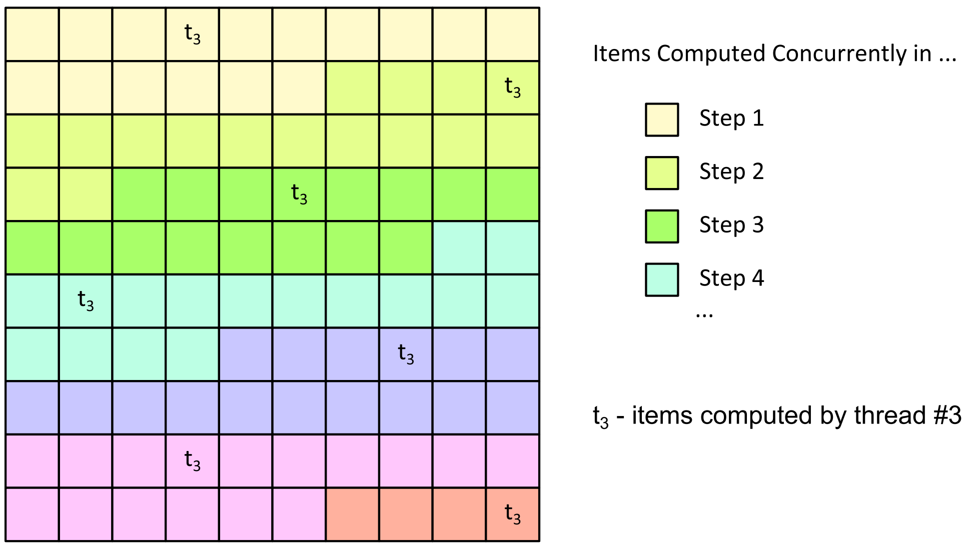

t3
t3
t3
t3
t3
t3
t3
Items Computed Concurrently in ...
Step 1
Step 2
Step 3
Step 4
...
t3 - items computed by thread #3

Figure 4.12: Dividing similarity matrix elements amongst the SMP threads (visualized for 16 threads)

can be dedicated to sum the partial results or a standard reduction tree can be used. Even though the tree reduction has time complexity $\mathcal{O}(\log T)$, it did not provide a measurable improvement, since all the threads must synchronize on a barrier after every step and the final summation is only a very small fragment of the overall work.

4.5 Parallel Approach to Metric Indexing

In the previous section, we have proposed a parallel implementation of SQFD for GPUs. The framework was designed to be used with any type of metric access method and we have tested it with pivot table prefiltering (2-phase LAESA). It was empirically observed, that the prefiltering step may not be sufficiently fast to supply blocks of candidates to GPUs. Especially if multiple GPUs are used and more expensive prefiltering is used, like the Ptolemaic prefiltering for instance.

4.5.1 Problem Analysis

To better understand the problem, we have studied how the effectiveness of the prefiltering step during standard kNN query evaluation on larger database. In this analysis, we hold to an assumption that the database is randomized. This assumption is quite realistic for pivot table indexing methods and we focus especially on the pivot table prefiltering. Figure 4.13 presents the prefiltering effectiveness ratio based on how large portion of the database has been already processed. These data were collected in sequential version of the kNN algorithm with combined triangular and Ptolemaic pivot table prefiltering, where filtering range value was updated after each candidate.

The horizontal axis represent the number of database objects processed so far. The vertical axis represent the prefiltering ratio, which is computed as the number of candidates yielded for distance computation divided by the number of total objects processed.

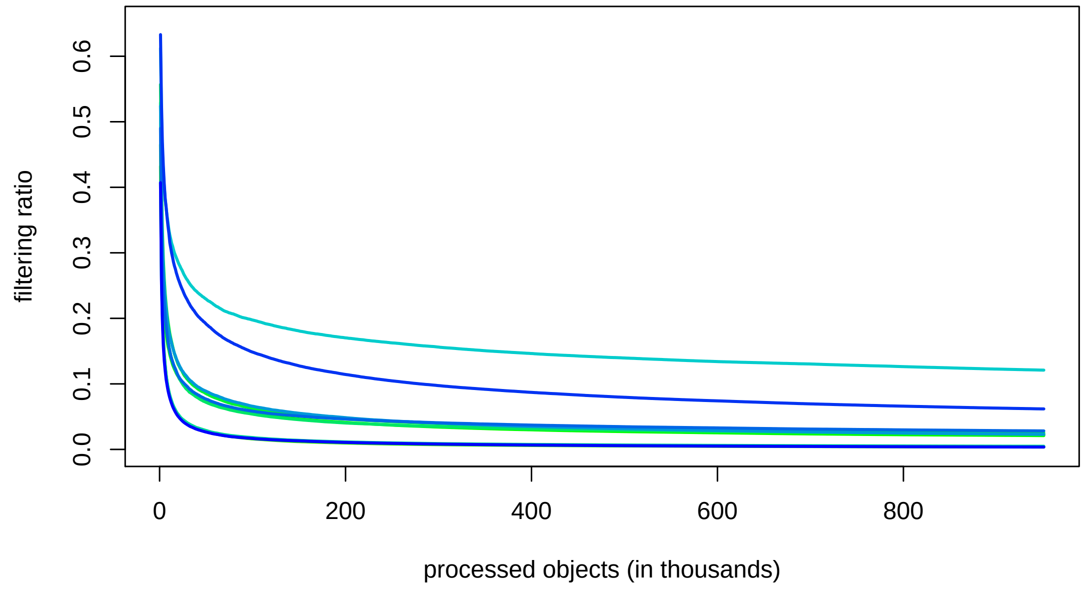


Figure 4.13: Pivot table prefiltering effectiveness of the kNN query performed on CoPhIR image database (10 randomly selected query objects, $\alpha = 0.2$)

As the data clearly show, most of the queries get close to the filtering range of 0.01 quite soon. This prefiltering range means, that at most one object of 100 makes it to the candidate block in average. If 32 pivots are used for Ptolemaic inequality, the prefiltering step must compute $|P|(|P|-1)/2 = 32 \cdot 31/2 = 496$ elementary lower bounds[4] to compute final lower bound for one object and $49,600$ elementary lower bounds to get one positive candidate. Even if we employ early termination optimization, which terminates the prefiltering immediately when a pivot is found that prunes the object out, (as described at the end of Section 4.2.2), the workload required to test one candidate on CPU starts to compete with the workload of the one distance computation on the GPU.

Possible Solutions

Both prefiltering and filtering steps are executed in turns by the main thread loop. The prefiltering step must be executed on CPU as it limits the number of signatures transferred to the GPU. If we moved the prefiltering to GPU, the entire database would have to be transferred to the GPU in blocks as if the sequential scan is being performed. In such case, the data transfers dominate the computational workload significantly.

Since the prefiltering must remain on the CPU, multiple cores could be used to accelerate the problem. As though this might help in some situations, such as the range query, it is definitely not suitable for kNN queries. A kNN query requires constant feedback between filtering and prefiltering as the partial top-k result, which is modified by the filtering step, provides a filtering range value for

[4]One elementary lower bound comprises a few elementary float number operations.

the prefiltering step. When divided into separate threads, a synchronization of the top-k result will be required. We have estimated that this synchronization would either create a serious bottleneck or the prefiltering step would be far less effective.

For these reasons, we have decided to keep both prefiltering and filtering steps in the main thread and accelerate the prefiltering by precomputing the lower bounds on GPU.

4.5.2 Precomputing Lower Bounds

To reduce the amount of work performed in the prefiltering step, the lower bounds for all database object are precomputed on the GPU(s) before the query is evaluated. In order to do so, the early termination optimization must be sacrificed and the implementation of the prefiltering step falls back to the condition $lb_P(q, c) \leq r$, where r is the filtering range and lb_P is the combined lower bound for pivot set P.

The 2-phase LAESA algorithm is extended into 3-phase version. The first phase remains the same – distances from query q to all pivots in P are computed. The new second phase computes lower bounds of all objects in the database. The third phase performs the same work as in original algorithm, only the prefiltering step is much cheaper thanks to precomputed lb_P values.

Lower-bounds on GPU

The computation of lower bounds on the GPU is very straightforward. Each lower bound is computed as the minimum of $|P|(|P|-1)/2$ values and two lower bounds can be computed independently, thus it is a perfect data parallel task. Each lower bound $lb_P(q, o)$ computation require only $|P|$ distances between query and pivots, $|P|$ distances between examined object o and pivots, and pivot-to-pivot distance matrix. Since query-to-pivot and pivot-to-pivot distances are shared amongst the threads, these data easily fit the local memory of an SMP. Therefore, one lower bound can be computed by one thread.

Threads in a group start their work by a cooperative load of shared data (query-to-pivot distances and pivot distance matrix) and corresponding part of the pivot table to the local memory. When the load is completed all the threads synchronize on a barrier and then continue with their own work.

If multiple GPUs are used, the total amount of work can be divided amongst them. In case the GPUs are of the same type and speed, the work may be divided evenly. Otherwise a speed estimation model would be required and the work should be divided in the ratio of the speeds of GPUs.

Pivot Table Representation

The only complication is finding an efficient representation of the pivot table as there are several requirements imposed.

- The pivot table must be easily divisible, so that each thread group can copy only the part required by its member threads.

- The pivot table must be organized, so it can be divided amongst multiple GPUs in an arbitrary ratio.

- The data must conform to the properties of the GPU memory, especially the banking mechanism of local memory.

Unfortunately, both row-wise and column-wise representations do not satisfy all the requirements. The row-wise approach is better for the pivot table partitioning. As the distances from each pivot to one object are gathered in a compact block, the data required by one thread group are in one continuous range, so they can be easily copied. Furthermore, the pivot table can be easily divided amongst the GPUs when necessary. On the other hand, this approach will cause many bank collisions in the local memory. Two consecutive threads access distances of two consecutive objects of the same pivot at the same time. This memory access is strided as these values are aligned to the number of pivots. If the number of pivots and the number of threads in warp have a common divisor larger than 1, bank conflicts are inevitable.

The column-wise approach has worked for us well in the past situations. The data are reorganized, so that distances from all objects to one pivot are compacted together. Therefore, the memory access pattern of the threads in a group will guarantee, that there will be no bank collisions what so ever, no matter how many pivots we have. On the other hand, it is much harder to divide the pivot table as the data required by one thread group or one GPU are fragmented into $|P|$ continuous blocks.

We create a compromise between both approaches called *packed pivot table* representation. The format is visualized in Figure 4.14. The pivot table is fragmented in row-wise manner into blocks of size B and within each block, the data are organized in column-wise manner.

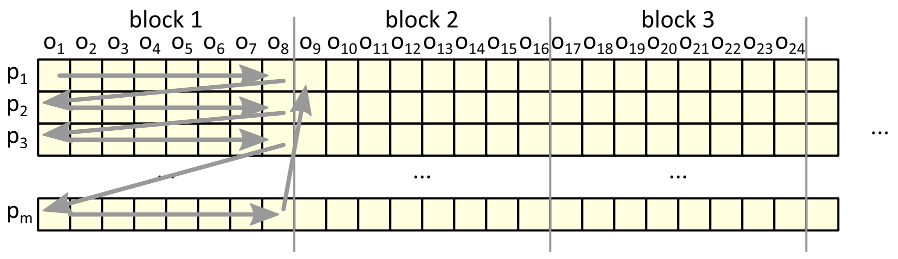

block 1
block 2
block 3
o1 o2 o3 o4 o5 o6 o7 o8 o9 o10 o11 o12 o13 o14 o15 o16 o17 o18 o19 o20 o21 o22 o23 o24
p1
p2
p3
pm
...

Figure 4.14: Packed pivot table representation

The packed pivot table can be easily divided with the granularity of the block size B, thus smaller sizes are prefered. The size of the block B must be a multiple of the warp size for obvious reasons. Considering the requirements, we have chosen block size of 256 as it is also the size of the work group, so each thread loads exactly one object (per pivot) from the block. The internal column-wise organization guarantees no bank collisions, each work group loads exactly one data block, and 256 is small enough for a fine grained division of the pivot table amongst multiple GPUs.

Persistent Pivot Table

An iterative approach can be chosen to compute the lower bounds, as it was used for SQFD. However, the pivot table is much smaller than signatures. If 32 pivots are used, each database object requires 128 B of space in the pivot table. Current Tesla GPU cards contain 6 GB of global memory, thus they can keep a block of pivot table for 48 million objects.

In case of smaller databases (tens of millions of objects), the pivot table can be kept persistently in the memory of the GPUs along with the buffers required for computing SQFDs in iterative manner. A comparison of signature sizes and pivot table record sizes leads to an assumption that if the object signatures can be cached in the host memory, the pivot table can be cached in the memory of GPUs.

4.6 Prefiltering for k Nearest Neighbours

The sequential scan algorithm for both range queries and k nearest neighbour queries can be easily parallelized. However, when a metric indexing is combined with the kNN query, a serious data dependency is created that prevents efficient parallelization. The prefiltering step requires the filtering range value which is updated by the filtering process as the partial top-k result is being updated.

This dependency renders the optimal prefiltered kNN algorithm serial in nature. If multiple distances are being computed concurrently, the update of the filtering range will be inevitably delayed. Some of the distances may be computed in vain as their corresponding objects could be pruned by the prefiltering if the filtering range was updated without delay.

This issue brings us a serious dilemma as the parallelism and the prefiltering effectiveness goes against each other. If the distance function is extremely expensive, it may be better to parallelize only the internal computations of the distance [108]. On the other hand, models with cheap distance functions or databases with high intrinsic dimensionality may not benefit from the prefiltering at all as it might be faster to compute a few cheap distances in highly parallel fashion than delay the candidate dispatching with prefiltering.

4.6.1 Block Dispatching Strategies

In our work, we have been working strictly with the SQFD, which stands somewhere in the middle from the perspective of our dilemma. The prefiltering is definitely improving the query evaluation performance. On the other hand, computing a few more distances in the multi-GPU setting is hardly measurable. Hence, we need to find a balance between the benefits of concurrent execution and prefiltering.

As it is quite hard to accurately predict the results of different strategies, an empirical approach was taken. We have implemented several strategies and observed the measured times for the same database and query set. Here, we would like to summarize our experience gained from the experiments.

Fixed Block Size and Limit

The simplest solution is to generate candidate blocks of *fixed size* and impose a limit on the number of pending blocks being dispatched to the GPUs. The strategy is summarized in Algorithm 4.8. It is is configured by two parameters: the B_{size} is the size of the blocks being dispatched to the GPUs and the B_{count} is the maximal number of pending blocks.

```
Data: database $D$, query $q$, $k$, block size $B_{size}$, pending blocks limit $B_{count}$
Result: list $R$ of the $k$ closest objects (and their distances)
$R \leftarrow \emptyset$, $B \leftarrow \emptyset$, $C \leftarrow D$
while $C \neq \emptyset \vee$ a pending block exists do
    if $C = \emptyset \vee \#$ pending blocks $\geq B_{count}$ then
        wait for next block of distances
    end
    if block of distances $d$ is waiting in output queue then
        filter $d$ to update $R$
    else
        select object $o \in C$, $C \leftarrow C \setminus \{o\}$
        if $o$ passed prefiltering then
            $B \leftarrow B \cup \{o\}$                    // add a new candidate
            if $|B| \geq B_{size} \vee C = \emptyset$ then
                dispatch $B$ to feeding threads
                $B \leftarrow \emptyset$
            end
        end
    end
end
```

Algorithm 4.8: Fixed block size and limit strategy

The algorithm prefilters the candidates and assemble them into block B. When the block reaches critical mass of B_{size}, it is dispatched to the feeding threads. The loop checks for the number of pending blocks and if it reaches the B_{count} limit, the main thread is suspended until a block of computed distances appears in the output queue of the GPU part. The filtering step is invoked whenever there is a block of distances ready, so the filtering range is updated as soon as possible.

The only remaining thing is to find optimal combination of B_{size} and B_{count}. The empirical results indicate that optimal B_{count} is $2 \cdot G$, where G is the number of available GPUs. The block size depends on some other factors. The data are summarized in the experimental section 4.7.3.

Adaptative Block Sizes

A natural extension of the previous strategy is to vary the B_{size} and B_{count} during the computation. The basic outline remains the same as in Algorithm 4.8. The only difference is that the two parameters can change in every iteration based on a predefined pattern, heuristic function, or some monitored properties of the query evaluation.

The idea behind this approach is that the prefiltering effectiveness increases as the top-k result gets closer to the query. Thus, at the beginning, the algorithm may benefit from more frequent updates of the filtering range, so smaller blocks are dispatched to the GPUs. As the prefiltering gets better, the size of the blocks can get larger, so the dispatching overhead is reduced.

We have tried several variations, but all of none of them was faster than the optimal fixed-size strategy. We strongly suspect, that this approach may work, but the block size changing pattern must be tailored to the dataset and to the query, which is extremely difficult to achieve.

Two-phase Approach

Inspired by the previous strategy is a *two-phase approach.* The idea that inspired adaptative approach is based on the fact that the prefiltering works poorly at the beginning of the query evaluation and improves rapidly as the intermediate top-k result is perfected. It was difficult to predict the pattern how the block size should change, so we simplify the situation.

The search is conducted in two phases. The first phase process only a small portion of the database, first several thousand objects for instance. In this phase, no prefiltering is used, so all the distances are computed. This should not be much worse than the previous strategies as the prefiltering does not do much good in the start. On the other hand, the absence of prefiltering unties our hands in matters of parallelism, so we can dispatch all the distances of the first phase at once to be computed concurrently.

In the second phase, the algorithm reverts to the original prefiltering with fixed-size blocks. The theory was that finding a switch point between first and second phase should be much easier than finding a (potentially very) complex pattern of the block size variation. Unfortunately, the empirical results were still below expectations. On the other hand, these results provided another insight to the problem, which was used to create range estimation algorithm presented in the following section.

4.6.2 Range Estimation Algorithm

Let us define an *optimal range* r_{opt} for a given query q and parameter k as a filtering range for the range query algorithm that produces result set $R_{r_{opt}}$, which is equal to the result set R_{kNN} produced by the kNN algorithm for the same query. If we knew the r_{opt} in advance, we could easily convert the kNN algorithm to the range algorithm, which would compute significantly less distances, and which would be perfectly parallelable.

Since we do not know r_{opt} in advance, we propose a *range estimation* algorithm. It computes a *range estimate* $r_{est} \geq r_{opt}$, which is then used for processing a range query. The result of the range query $R_{r_{est}}$ is obviously a superset of the kNN result since $r_{est} \geq r_{opt} \Rightarrow R_{r_{est}} \supseteq R_{r_{opt}} = R_{kNN}$. The final kNN result can quickly be extracted from the $R_{r_{est}}$ by standard filtering as the distances have already been computed.

The Range Estimation

The most essential part of the algorithm is the range estimation process. The algorithm selects a *range estimation set* $E \subset D$ of predefined size $|E| = E_{size}$. This set is used in the first phase of the algorithm where standard kNN query without any prefiltering is resolved. We assume that the $E_{size} \geq k$, so we can take the distance of k-th item of the result as estimate range r_{est}.

The quality of the range estimate is affected by the selection of estimate set E. We need to find a cheap way of selecting objects for E that gives us the best possible estimate. Our approach is based on the observation that the filtering works significantly better, if the database objects are sorted by their lower bounds and processed in ascending order. Therefore, E_{size} objects with the smallest lower bounds of D should be selected in set E (i.e., $\forall e \in E, \forall o \in D : lb(e) \leq lb(o)$).

The sorting process takes a significant amount of time, even if performed on the GPUs. Fortunately, we do not need to sort the database to select top E_{size} objects based on their lower bounds. The same approach which uses the kNN filtering step can be employed here. A 2-regular heap with limited size of E_{size} is created and all the lower bound values are inserted into this heap, so the heap ends up with E_{size} smallest lower bounds and their respective objects. This step can be accelerated further by the GPU as the lower bounds are computed there anyway, but the code profiling indicate that this step is so cheap, it can be performed on CPU without causing any measurable drop of performance.

Algorithm Formalization

The range estimation approach is formalized in Algorithm 4.9. The first phase of the algorithm, which estimates the filtering range, has been already described. The most time of the first phase is spent by distance computations of the kNN query, but these distances can be computed on the GPUs in highly parallel fashion since no prefiltering is used.

The second phase consist of a range query algorithm performed on the remaining objects of the database. The range query employs pivot table prefiltering, but with fixed range, the distances can be computed concurrently without any dependencies. The results of the range query can be easily integrated to the result set yielded by the first phase. In fact, this can be done on the fly without explicitly constructing the $R_{r_{est}}$ set.

```
Data: database $D$, pivots $P$, query $q$, $k$, estimate set size $E_{size}$
Result: list $R$ of the $k$ closest objects (and their distances)
compute $d(q,p)$ for each $p \in P$
compute lower bounds $lb_P$ on the GPU(s)
// range estimation phase
select $E \subseteq D$ of size $E_{size}$, so that $\forall e \in E, o \in D \setminus E : lb_P(q,e) \leq lb_P(q,o)$
$R \leftarrow k$NNQuery$(q,k,E)$                              // no prefiltering
$r_{est} \leftarrow \max\{d(q,o) | o \in R\}$
// range query phase
$R_{r_{est}} \leftarrow$ RangeQuery$(q, r_{est}, D \setminus E)$          // pivot table prefiltering
update $R$ with the results from $R_{r_{est}}$
```

Algorithm 4.9: The range estimation algorithm for kNN query

The algorithm is left with one parameter, that affects the performance. The optimal size of the estimate set depends on the properties of the database (such as intrinsic dimensionality), on the time complexity of the distance function, on the strength of available GPU devices, and on the query itself. We provide empirical results in the experimental section 4.7.4 that should give us at least some estimates for our datasets. This matter requires additional research, which is beyond the scope of this thesis.

The Misestimation Problem

The greatest (and perhaps the only serious) problem of the range estimation algorithm is the possibility of range misestimation. If the first phase yields a range estimate that is many times higher than the optimal range r_{opt}, the algorithm will not perform much better than sequential scan.

One of the possible solutions is to switch to kNN algorithm with the fixed block size strategy. This approach should not be worse than the original kNN algorithm, as the set E usually represents only a very small portion of the database and the prefiltering does not work very well in the start. The decision, whether to use range query or fall back to kNN query can be done in runtime. In order to do so, we need some evaluation mechanism which can predict the quality of the estimated range. This mechanism is a subject of future research.

4.7 Experiments

The experiments are divided into three parts that correspond to the contribution sections in this chapter. The first part evaluates the performance of our new SQFD implementation for GPUs. It is followed by experiments focusing on combining metric indexing with our parallel SQFD and accelerating the indexing by lower bounds precomputations on GPUs. Finally, we address the question of efficient parallel kNN query with pivot table prefiltering and the performance of newly proposed range estimation algorithm.

4.7.1 Hardware and Methodology

Before we present our empirical data, let us introduce the hardware setup, used datasets, and the methodology of measurement.

Hardware

The GPU experiments were performed on a server built on a special motherboard (FT72-B7015) designed to embrace up to 8 GPU cards. The server was equipped with Xeon E5645 processor comprising 6 physical (12 logical) cores running at 2.4 GHz, 96 GB of DDR3-1333 RAM, and 4 NVIDIA Tesla M2090 GPU cards based on Fermi architecture. Each GPU chip consists of 512 cores (32 cores per 16 SMPs) and 6 GB of memory.

We also tested the GPU implementation on a commodity PC with two gaming cards NVIDIA GTX 580. These cards have also 512 cores, but only 1.5 GB of memory. We have found that the GTX 580 cards have similar performance as the Teslas, thus we do not provide more detailed comparison.

Tests conducted on the multi-core CPU server platform are denoted CPU1 – CPU48 in the figures. We used Dell M910 server with four six-core Intel Xeon E7540 processors with hyperthreading (i.e., 48 logical cores) clocked at 2.0 GHz. The server was equipped with 128 GB of RAM organized as 4-node cache coherent NUMA. A RedHat Enterprise Linux 6.3 was used as operating system on all machines.

Data

We use three datasets in our experiments. All these datasets represent real world image sets, but they differ in the number of objects, sizes of the signatures, and indexability properties. The smallest one is The Amsterdam Library of Object Images (ALOI) [118], which consists of $72,000$ images. The ALOI database is too small for some experiments, but it can be used for sequential scan tests, which are quite slow.

A medium-size dataset is the CoPhIR [119], which is a wellknown dataset for content-based image retrieval testing. It was designed to test both effectiveness and efficiency of the image retrieval systems. Our subset comprises of $951,532$ randomly chosen images.

Finally, the largest set is the Profimedia [120]. Profimedia is a commercial image database available on the internet. It offers photographs that can be used for illustrational purposes in advertising, propagation, and many other domains. We have a subset of 17.5 million randomly chosen images.

All the datasets have been randomly shuffled and feature signatures were extracted for them. The shuffling was performed by Algorithm 4.10, where the random function is a pseudo-random generator implemented in standard C++ library. We do realize that this generator does not produce completely random numbers, but since we are not trying to ensure cryptographic safety, the pseudo-random numbers ensure sufficient shuffling.

```
**Data**: array $A$ of $N$ items (indexed from 1 to $N$)
**Result**: shuffled array $A$
**for** $i \leftarrow 1 \dots N-1$ **do**
    $j \leftarrow \mathrm{random}(i, N)$                    // $j \in \{i, \dots, N\}$
    **if** $i \neq j$ **then**
        swap $A[i]$ and $A[j]$
    **end**
**end**
```

Algorithm 4.10: The random shuffling algorithm

The database is shuffled for two reasons. First of all, we need to normalize our results as we would like to show, how our methods work with unknown query on any data. If the objects in the database use some specific ordering, it would be hard to prove that our experiments were not tuned for this particular ordering. Second reason is, that the pivot table prefiltering expects the objects to be randomized. Unlike other metric access methods, this prefiltering traverses the entire database, so it would be better if it has a chance to encounter any object with the same probability.

The first 32 objects were used as pivots for the pivot table prefiltering. We need to select random objects as pivots, but since the database was already randomized, the first 32 objects are as good selection as any. A subset comprising 100 objects was selected and excluded from the database. Objects in this subset were used only as testing queries. Again, we wanted to select query objects randomly, so we choose last 100 objects of the shuffled database.

Methodology

The tests were performed using 100 query signatures with different numbers of centroids. The time of evaluation was measured using the system real time clock. Each experiment was measured at least three times and the presented results are the mean values of the measured times. If any of the measured values deviated from the average by more than 15%, the value was discarded as tainted and the test was repeated.

The most of the presented values are the average times computed from 100 queries. The only exception are the box plots, where the distribution of all 100 times is presented.

4.7.2 Parallel SQFD

In this section, we evaluate the performance of standard sequential scan algorithm where the distances are computed concurrently on GPUs. All presented experiments resolve a kNN query, where $k = 100$. No range queries were presented as they exhibit almost identical performance results. The filtering steps of both query types consume similar amount of time and the most of the time is spent in the distance computations anyway.

Finding Optimal Block Size

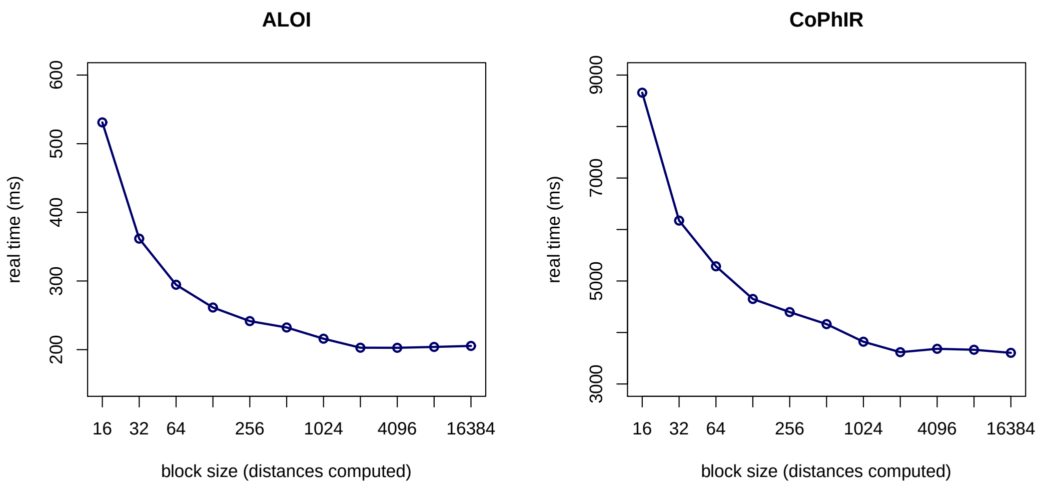


Figure 4.15: Impact of the block sizes on sequential scan (1 GPU)

First of all, we would like to find an optimal number of distances being computed at once in one block. We use single-query setup, thus for a block size B, $B + 1$ signatures are dispatched in each block and B distances are computed. There are always two blocks being dispatched to each GPU. One of the blocks is being processed, while the other is being transferred.

The results depicted in Figure 4.15 confirm our general assumption, that smaller blocks have higher overhead, thus processing the query is faster when the blocks are larger. The block size should be at least 2048 for both datasets. All the remaining SQFD experiments are performed with block size 4096 unless explicitly stated otherwise.

Self-similarity Precomputation

The SQFD can be optimized by precomputing the self-similarity matrices of all database object and the self-similarity matrix of the query object before the query evaluation starts. As shown in Section 4.4.3, each self-similarity matrix can be stored as one partial sum. We have tested the impact of this optimization on the performance.

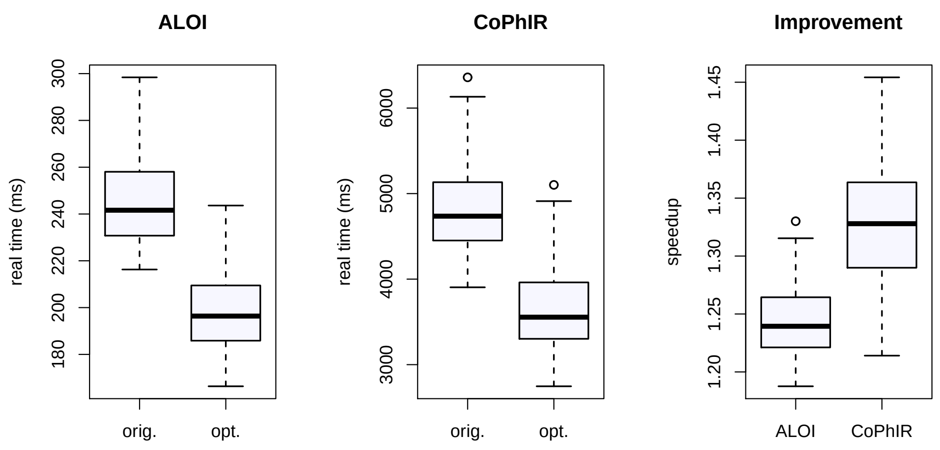


Figure 4.16: Impact of the self-similarity precomputation (1 GPU)

Figure 4.16 presents the measured improvement in box plots, which demonstrate also the distributions of times. The data denoted *orig.* represent times for the SQFD implementation, which precomputes the query self-similarity matrix so it is not computed repetitively with every distance, but the self-similarity values of the database objects are not precomputed. The *opt.* results have all self-similarity values precomputed, thus each distance enumerates only the inter-similarity matrix. The *improvement* graph shows the speedup values for individual queries computed as $T_{orig.}/T_{opt.}$ ratio.

We can observe, that the speedup of *opt.* version varied significantly over the query set. This is caused by different sizes of the signatures which affects the ratio of precomputed work to total work. Furthermore, the CoPhIR database has larger signatures (according to histogram in Figure 4.9), thus more work is saved when the self-similarity matrices are precomputed.

Sequential Scans on Various Hardware Configurations

To demonstrate the impact of GPU acceleration, we have performed series of queries with optimal settings on several platforms. The CPUn denotes experiments running on our NUMA server comprising 4 processors, 6 physical cores with hyperthreading each. The CPU implementation used Threading Building Blocks managed thread pool with n threads. The SQFD was compiled to exploit SSE instruction, so the CPU was utilized to the best of its abilities. The GPUn tests denote the results for GPU implementation restricted to n devices.

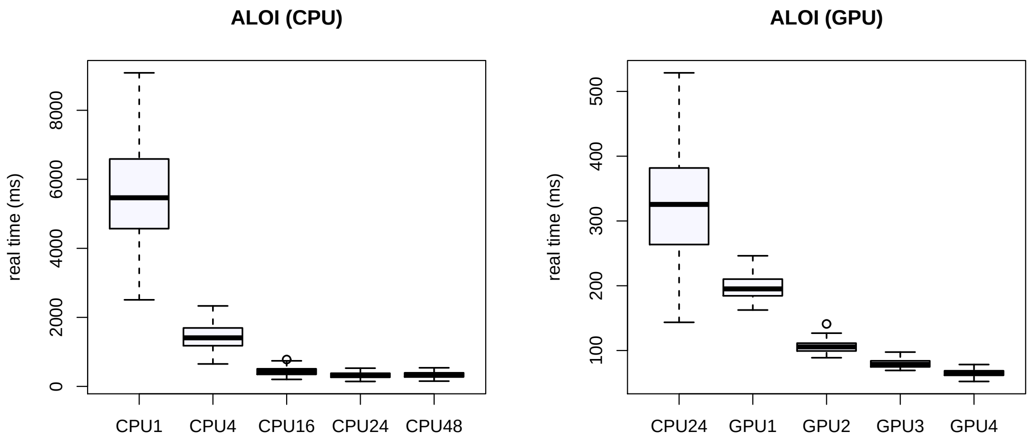


Figure 4.17: Results of 100NN queries on ALOI database for various architectures

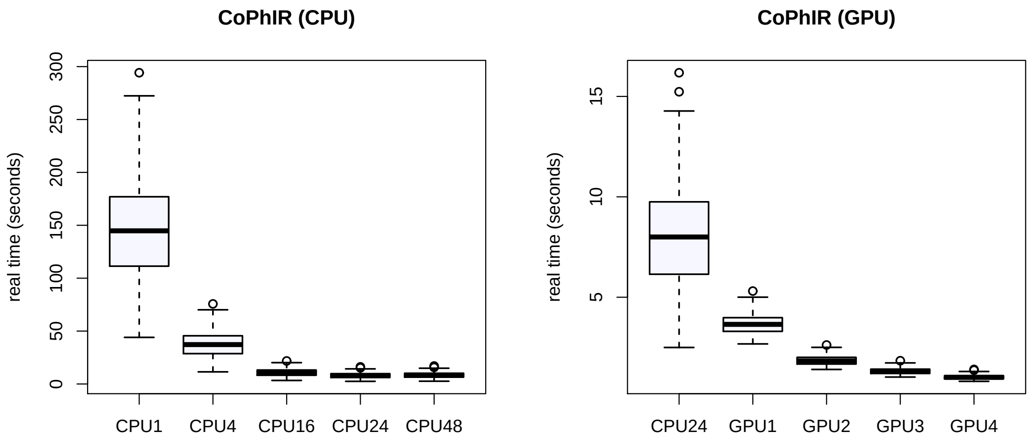


Figure 4.18: Results of 100NN queries on CoPhIR database for various architectures

Figures 4.17 and 4.18 presents the results measured on ALOI and CoPhIR datasets respectively. The CPU results revealed interesting anomaly, as the CPU24 version was faster than CPU48 version. The server has 24 physical and 48 logical cores. Hyperthreading technology has obviously its limits and the workload of SQFD is capable of utilizing most of the CPU internal units. When two threads are competing for the resources of the physical core, no additional performance is gained and the overhead of the parallel execution causes slight drop in performance.

The speedups relative to the single core version are shown in Figure 4.19. The GPU speedups are in fact comparing two completely different platforms and they should be perceived in such context. However, they give us a general idea, how much the performance is improved by employing GPUs for similarity search.

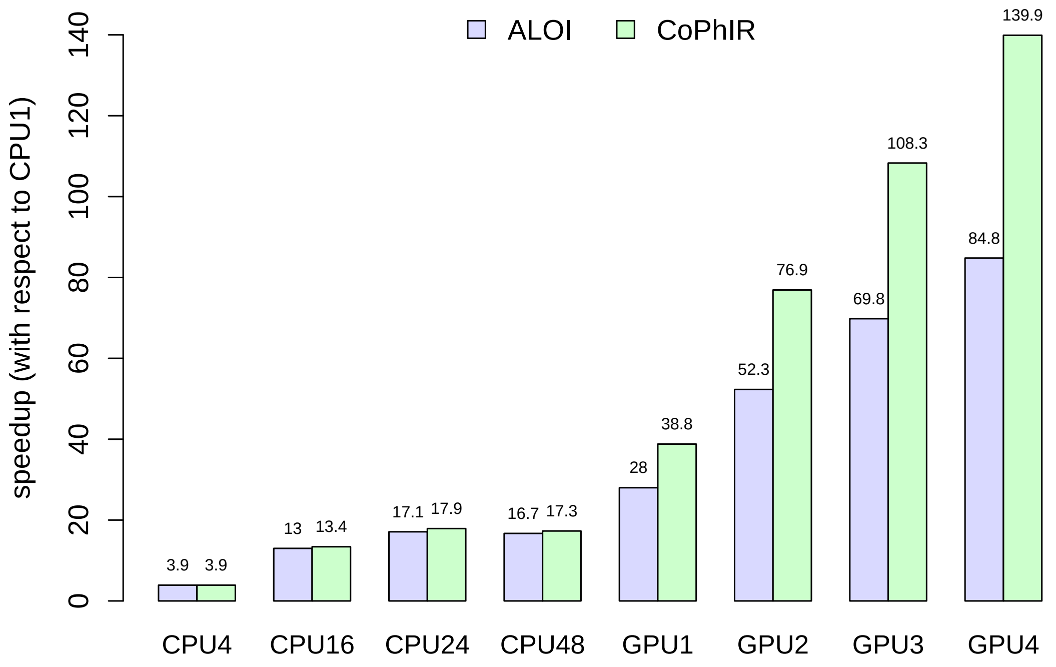


Figure 4.19: Speedup to the serial version for various platforms

The ALOI database exhibits significantly lower speedup than the CoPhIR dataset. This is most likely caused by the data properties of ALOI as it comprises only $72,000$ objects. Hence, the GPUs are provided only few signature blocks each thus the pipeline effect is not exploited to its full potential. Furthermore, the ALOI has smaller signatures, thus the ratio of computational work to data transfers is significantly lower than in case of CoPhIR.

4.7.3 Metric and Ptolemaic Indexing

We have proven that GPUs can really help accelerate distance computations. Now we focus on metric access methods, since they are often used in combination with SQFD.

The Alpha Parameter

First of all, we would like to demonstrate the impact of the SQFD parameter α on the indexing effectiveness. The Figure 4.20 shows the results of 100NN query with triangular pivot table prefiltering using different values of α. The fixed block size strategy was used to deal with the dependency problem of the parallel kNN queries that employ prefiltering. The block size determines the number of distances being computed in one block and at most 2 blocks per GPU are dispatched simultaneously.

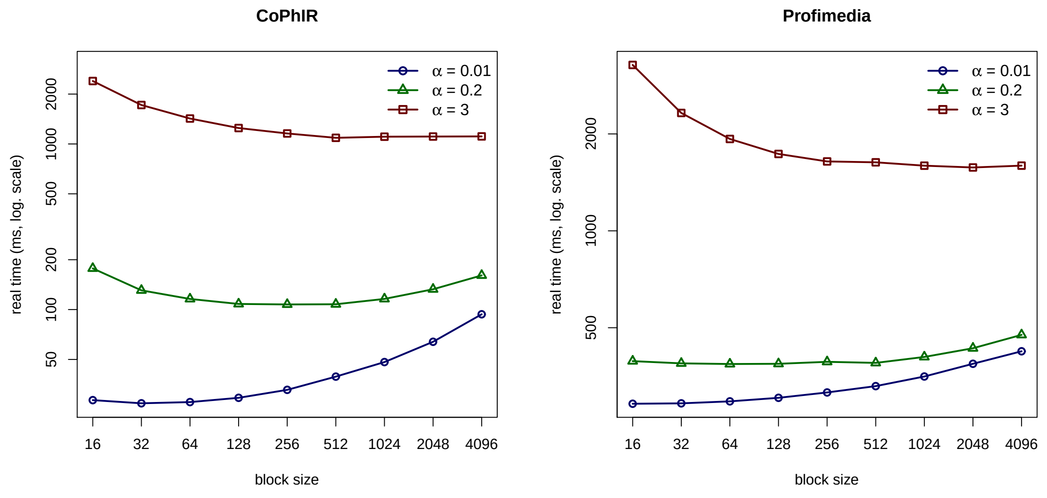


Figure 4.20: Fixed block size 100NN query with triangular pivot table prefiltering performed on 4 GPUs

The values $\alpha = 0.01$, 0.2, and 3 were selected according to previous research [121]. The $\alpha = 0.01$ exhibits the best indexability properties, $\alpha = 3$ has the best similarity precision, and $\alpha = 0.2$ is the best compromise between indexability and precision.

As expected, the indexability affects the performance significantly. The lowest α causes that more objects can be pruned in the prefiltering phase, thus less distances are computed. On the other hand, the $\alpha = 3$ does not seem to benefit from the prefiltering at all as it exhibits similar results to a simple sequential scan algorithm (Figure 4.18).

The optimal block size also depends on α. When $\alpha = 0.01$, smaller blocks work better as the filtering range is updated more often. Slightly larger blocks (from 256 to 512 objects) are required for $\alpha = 0.2$, since the balance between index effectiveness and GPU distance computations overhead shifts with the decrease of the indexability.

Precomputing Lower Bounds on GPUs

We have proposed to compute the lower bounds on GPU(s) in order to accelerate the prefiltering step and achieve faster dispatching of candidates. The impact of the precomputation is assessed independently for all types of prefiltering using 1 and 4 GPUs on CoPhIR database (Figure 4.21) and Profimedia database (Figure 4.22). Method *tri* denotes standard triangular prefiltering, *pto* stands for Ptolemaic prefiltering, and *cmb* is combined (triangular and Ptolemaic) prefiltering. The versions suffixed with asterisk (*tri**, *pto**, and *cmb**) precompute their lower bounds on GPU(s) and employ 3-phase LAESA access method. Let us emphasize, that the methods with precomputed lower bounds are timed so that all phases (including the precomputation) are encompassed in the measured time.

All methods use 32 pivots. Let us revise that triangular prefiltering use one pivot for each individual lower bound estimate, thus computing the $lb_P^{\triangle}$ has time complexity $\mathcal{O}(|P|)$. Ptolemaic prefiltering lb_P^{Pt} uses a pair of pivots for each lower bound, hence its time complexity is $\mathcal{O}(|P|^2)$.

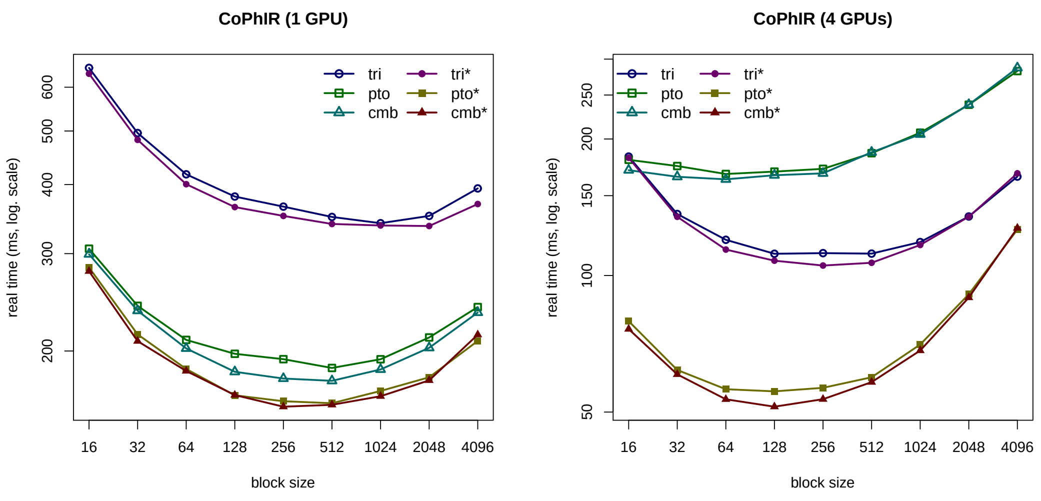


Figure 4.21: Fixed block size 100NN query with various types of pivot table prefiltering on CoPhIR database ($\alpha = 0.2$)

The CoPhIR database results confirm our assumptions. When one GPU is used, we can observe two important things. First of all, the Ptolemaic prefiltering is more effective on the same set of pivots as it can give us better lower bound estimates and combined prefiltering is the best as it yields the better of the lower bounds produced by previous two methods. Furthermore, the lower bound precomputation can save some time as *tri**, *pto**, and *cmb** versions are faster than *tri*, *pto*, and *cmb* respectively.

The situation gets slightly different as more parallel computational power is available. The four-GPU version exhibit strange behaviour, since the triangular prefiltering on CPU outperformed both Ptolemaic and combined prefiltering. The explanation of this anomaly is apparent, when we compare the prefiltering methods which have the lower bounds precomputed. Without precomputation, the CPU cannot supply candidates to GPUs fast enough. Hence, the better performance is achieved, when cheaper (triangular) prefiltering is employed. The triangular prefiltering causes that more distances are computed by the GPUs, but the GPUs do not get stalled by the main CPU thread. When we precompute the lower bounds, the combined prefiltering method is the best, since it prunes more objects and fast prefiltering does not stall the GPUs.

The results measured on Profimedia database also confirms that the prefiltering with precomputed lower bounds is better. However, the data exhibit some

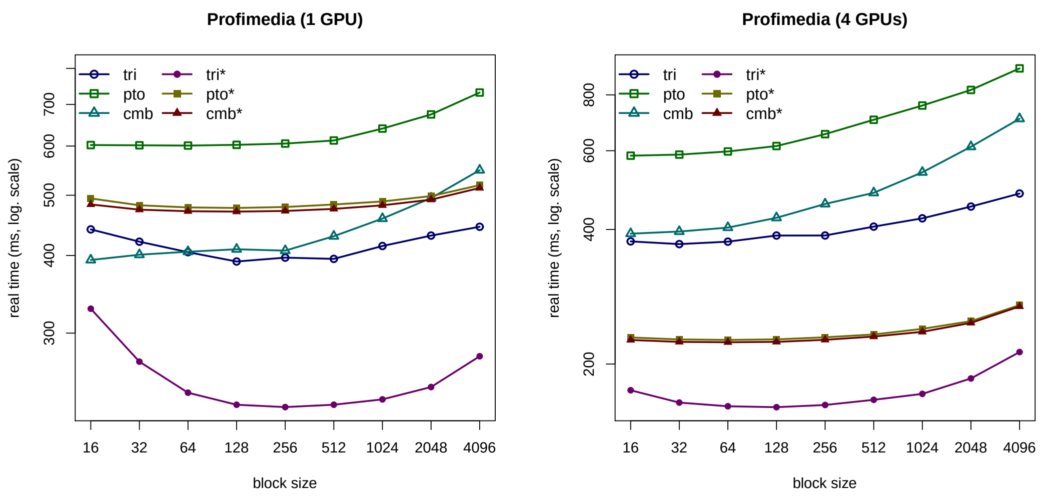


Figure 4.22: Fixed block size 100NN query with various types of pivot table prefiltering on Profimedia database ($\alpha = 0.2$)

other unexpected anomalies we need to explain. First of all, in almost every case, the triangular prefiltering outperformed Ptolemaic and combined prefiltering, even when the lower bounds are precomputed. The reason is that the Profimedia database has better indexability than CoPhIR, thus even simple triangular inequality is sufficient to rule out many objects in the prefiltering step. The Ptolemaic prefiltering is much more expensive and it improves the pruning only slightly. Therefore, it is better to use cheaper prefiltering in this case.

Second important observation is that the combined prefiltering with precomputed lower bounds is slower than the CPU prefiltering. This behaviour is caused by the early termination optimization that is employed on CPUs. We have already established, that the triangular prefiltering works very well on Profimedia dataset. If the prefiltering of an object is terminated as soon as a pivot (or a pair of pivots in case of Ptolemaic inequality) is found that gives us a lower bound exceeding the filtering range, a lot of computational work will be saved.

The precomputation of 17.5 million lower bounds on one GPU takes about 75 milliseconds in case of triangular inequality, around 375 milliseconds for Ptolemaic inequality. If the CPU prefiltering becomes cheap enough so it overlaps significantly with the distance computations and does not stall the GPUs, the overall performance may be better than in case a long time is spent by precomputing the lower bounds on GPU(s).

4.7.4 Parallel k Nearest Neighbours with Prefiltering

Finally, we address the problems of parallel kNN queries with pivot table prefiltering. So far, we have presented only the results of fixed block size approach. Now, we examine the range estimate algorithm, which should outperform the traditional fixed block size approach in most cases.

Size of the Range Estimation Set

First of all, we need to determine optimal estimate set size. For this purpose we have measured the performance of the range estimate algorithm with E_{size} between 128 and $16,384$. Since we test kNN queries where $k = 100$ and E_{size} is required to be greater than k, we did not consider any smaller sets. The performance starts dropping significantly, when the estimate set exceeds the size of $16,384$, thus we did not consider to include larger sets either.

To see how the E_{size} parameter depends on other properties, we conducted the experiment on one and four GPUs using α value of 0.01 and 0.2. It has been established that $\alpha = 3$ does not benefit from indexing, so we did not include it to this experiments. All the experiments were using combined triangular and Ptolemaic prefiltering method.

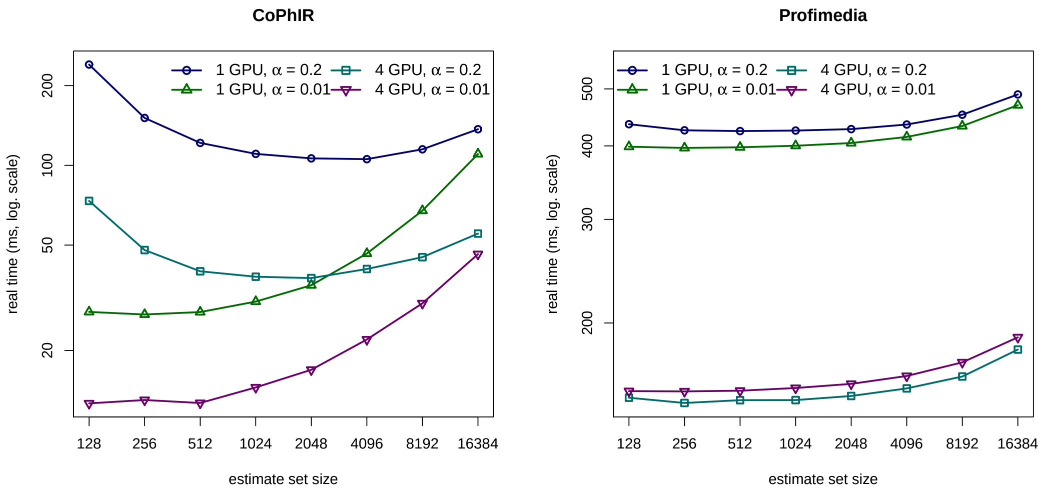


Figure 4.23: Finding optimal E_{size} for different databases and configurations

The results presented in Figure 4.23 match the observations from the fixed block size approach (Figure 4.20). Smaller estimate sets (up to 512) are better for $\alpha = 0.01$, as it has the best indexability. If we change α to 0.2, slightly larger estimate sets are optimal.

The Profimedia database exhibit much lower variance in the performance across the E_{size} parameter domain. This is caused by the same problem, that was observed in the previous section. The range estimation algorithm needs to precompute all the lower bounds to select the estimate set. The precomputation itself takes significant portion of time in case of Profimedia, and this time depends mostly on the size of the database.

It has been also observed, that the optimal estimate set size is different for each query. The data presented in Figure 4.23 are computed as average time for 100 randomly selected queries, so we can assume that the optimal sizes found in the graph work well in average. However, it might be better to select the estimate set size based also on some preliminary information about the query. This research is still to be concluded.

Overall Comparison

To assess the benefits of the range estimation algorithm, we compare it with the fixed block size approach. On the following graph (Figure 4.24), *fixed* denotes the fixed block size method, *range* denotes the range estimation algorithm, and *divine* denotes the optimal range estimation algorithm. The optimal range estimation algorithm uses oraculum to guess the optimal filtering range and then performs parallel range query as a substitution for the kNN query. The divine method is obviously theoretical, but it provides a baseline as the kNN query with pivot table prefiltering cannot be resolved any faster.

The results presented in Figure 4.24 are the average times of 100 queries performed with the optimal parameters (block sizes or estimate set sizes) determined for each configuration independently by the previous experiments. Combined prefiltering method was used in the experiments.

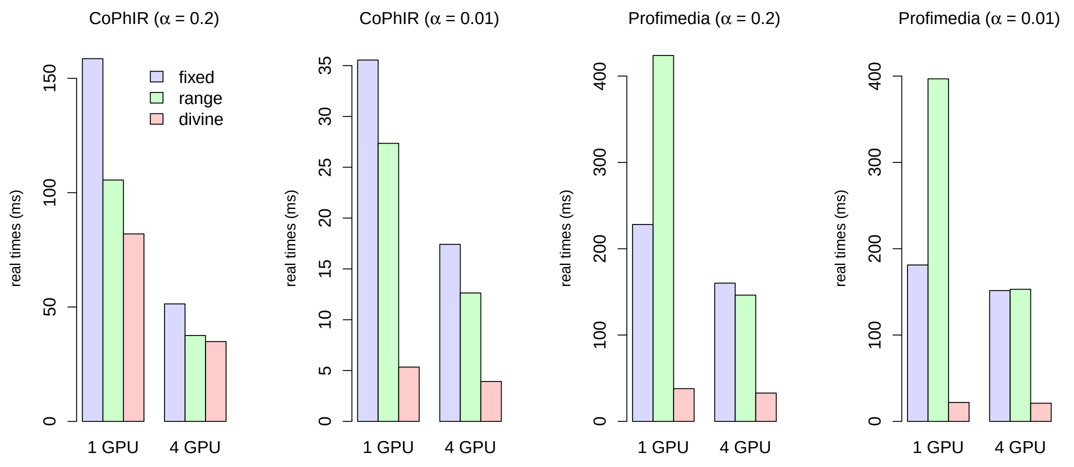


Figure 4.24: Comparison of presented algorithms for kNN queries with prefiltering

In case of CoPhIR database, the results clearly prove, that the range estimation algorithm outperforms the fixed block size approach and for $\alpha = 0.2$ it gets quite close to the results of the divine algorithm. The Profimedia dataset exhibits the anomaly described earlier. The range estimation algorithm needs to compute all lower bounds on the GPU(s). In one-GPU configuration, this precomputation takes 375 milliseconds, which alone is much longer than the time required by the fixed block size approach.

5. Image Feature Extraction

In the previous chapter, we used the GPUs to accelerate the content-based retrieval methods on the image databases. In this chapter, we focus on another computationally expensive problem of the similarity search – the construction of object descriptors. The images are represented by the feature signatures, which can be easily compared by the SQFD function to test their dissimilarity. The feature extraction process that constructs these signatures is quite time consuming. We propose a GPU accelerated extractor, which can speedup this process by two orders of magnitude.

5.1 Introduction

This chapter continues the work on similarity search problems presented in the previous chapter. The image feature signatures have been introduced in Section 4.1.2. We revise the feature signatures in more detail and present the feature extraction process in Section 5.2. The performance issues and our GPU accelerated implementation are addressed afterwards in Section 5.3.

5.1.1 Motivation

The efficiency of feature extraction could seem little of importance, since the extraction process is usually performed only once for each image. The images are processed independently, so multiple extractions can be performed concurrently, even on separate machines. Hence, we can achieve desired extraction throughput by purchasing enough hardware or computational time in a cloud. However, there are several reasons, why we should consider using GPUs for the extraction:

- We have already employed GPUs for similarity search, so we can assume that the GPU devices are available for the database system. It might be beneficial to utilize their computational power for all computationally expensive problems.
- The GPUs have much better performance to cost and performance to power consumption ratios. Therefore, it may be cheaper to acquire sufficient computational power employing GPUs, instead of traditional multi-core CPUs and NUMA systems. Analogically, we can process larger datasets in feasible time with limited hardware budget.
- The extraction algorithms has many configuration parameters. Finding an optimal configuration via experimental approach requires many iterations of the extraction process and subsequent precision verification. If we reduce the time required by the extraction process, we can explore later parameter space, or even switch to dynamic configuration model which tunes the parameters specificly for each dataset.

Presented reasons are emphasized in the light of our research and related projects. The GPU extractor reduced feature extraction time of large datasets that we require for related experiments from matter of weeks to matter of hours. Furthermore, we were able to perform quite extensive search of optimal parameter configuration on a database with ground truth in matter of weeks, while the same experiments would take more than a year on common multi-core CPU.

5.1.2 Related Work

Several types of image descriptors were already introduced in Section 4.1.2. The extraction algorithm for feature signatures consists of feature sampling and clustering, so we focus on related work in parallel image processing and GPU implementations of the k-means clustering.

GPU Image Processing

Employing GPUs for image color processing was suggested first in the work of Colantoni et al. [122] in 2003. Among other things, they have utilized GPU cards of the time for color conversions from RGB to Lab and to HSV color spaces. They have used fragment shaders to compute the conversions and achieved 10$\times$ speedup with respect to single-core CPUs.

Methods of computer vision are employed to extract more complex features than colors. One of the first attempts to utilize GPU devices for this task was presented by James Fung in the book of GPU Gems [123]. He proposed using GPU shaders to accelerate several algorithms for edge detection, hands tracking, or feature vector computations.

One of the most important problems in the computer vision and feature extraction is the edge detection. Perhaps the most famous algorithm was presented by Canny [124], so it is called *Canny Edge Detection* algorithm. A GPU implementation of this algorithm was proposed by Roodt et al. [125], in 2007. Their solution was based on graphical approach and implemented in GLSL language using OpenGL API. They have achieved a throughput of 80 frames per second on $2,048 \times 2,048$ images on one NVIDIA GeForce 8800 GTX, which is enough to perform realtime processing of HD video. Unfortunately, no detailed comparison with CPU performance was given.

Independently on the work of Roodt, similar paper was presented one year later by Luo et al. [126]. Their implementation used CUDA framework and achieved similar results on the same hardware. They have also implemented the hystersis step that post-process the detected edges. Two years later, Ogawa et al. claimed that the algorithm of Luo has a flaw as it does not correctly traverse all weak edge pixels, so they propose a correction of the algorithm [127].

Another type of feature signatures can be generated using Scale Invariant Feature Transform (SIFT) method. This method is usually used to detect objects by their shapes, since it is highly resilient to changes in rotation, scaling, and lightning conditions. Heymann et al. [128] proposed an implementation optimized for GPUs. They have used OpenGL to accelerate individual steps of the extraction, such as gradient transformations, gaussian convolution or key point filtering. They have achieved 6$\times$ speedup with respect to a single-core CPU.

K-means Clustering on GPUs

One of the first attempts to parallelize clustering was presented in 1989 by Xiaobo Li et al. [129]. They designed several parallel clustering algorithms for Hypercube SIMD computer model. Their approach was purely theoretical, but designed for an existing architecture.

The first attempt to accelerate k-means clustering on GPUs that we know of was made by Shalom et al. [130]. Their implementation did not use any established computational framework, but rather convert the clustering into image rendering problem. They have tried several approaches with the OpenGL platform, such as using the texture buffer or using the depth and stencil buffer. The shaders that perform the computations were written in GLSL and they performed multi-pass rendering to perform the iterative refinements.

Independently on the work of Shalom, Farivar and his team presented work [131], which used CUDA to compute the k-means. Their implementation was designed to accelerate the two most expensive steps – the cluster assignment and the computation of centroids.

One year later, another two papers on the subject was presented by Hong-tao [132] and by Zechner [133]. They have achieved slightly better performance, but the general idea of the algorithm remained the same in both papers.

All the presented work on GPU k-means clustering that we know of tries to accelerate one instance of k-means problems with explicit CPU-GPU synchronization after each step in the main loop. Our problem is different not only because we use modified version of k-means algorithm, but also because we need to solve clustering problem on multiple smaller datasets. Furthermore, no one has presented work that utilizes multi-GPU configurations yet, at least not to our best knowledge.

5.2 The Feature Extraction

The feature extraction process takes an image and produces its feature signature descriptor. We expect that the image is represented in some standard raster format, so we can access the color information of each pixel directly. The signatures produced by the extraction process were briefly introduced in Section 4.1.2. We revise the feature signature in more detail and explain, how they are extracted from the image.

5.2.1 Extraction Overview

The signature S^o is a set of features, which are points from feature space F_s. Our feature space is in fact a subset of $\mathbb{R}^7$, where each dimension has special meaning. A point $f \in F_s$ looks like $f = (x, y, L, a, b, c, e)$. The (x, y) coordinates represent a position within the image. This position is the center of a sampling area where the other features are taken. The (L, a, b) properties hold the color information of that area and (c, e) are the contrast and entropy values of the texture in that area.

Each feature f from the signature S^o is accompanied by a weight value $w_f \in \mathbb{R}^+$, that summarizes the importance of the feature within the set. In other words, how large portion of the image does the feature represent. A larger area with the same color and texture can be represented by a single feature with greater weight while more heterogeneous areas are represented by multiple features with smaller weights.

The features are determined by a clustering process, so we also denote them centroids. Since the features and weights are provided together in the signature, we can mend the original definition as follows. The signature S^o is a set of ordered pairs (f_i, w_i) (or (c_i, w_i)), where f_i are the features (or c_i are the centroids) and w_i are the weights.

The Extraction Process

The extraction process consist of following steps:

1. Image preprocessing
2. Feature sampling
3. Clustering

The image preprocessing can be used to change the size of the image or apply some graphical filters such as blur distortion. These procedures can be used to normalize the images in the database (e.g., to the same size and proportions) and to reduce information noise.

In our implementation, we have resized all the images into thumbnails of 150×150 pixels, which were saved as RGB bitmaps using 8-bits per channel. Even though the original images do not have rectangular size, this method is designed to operate on photographs or similar type of imagery, which can benefit from such normalization. No additional filters were applied.

The image preprocessing is not very interesting as the image resampling techniques, as well as filtering algorithms, are thoroughly described in the literature [134] and we do not address them in this work. The feature sampling and the clustering process are described in the following two sections.

5.2.2 Sampling Features

The entire picture contains too many information, so a sampling technique is used to select only some representatives. The sampling is used mainly to speedup the extraction process, but it can be also used to prioritize certain parts of the image if nonuniform distribution is selected.

To sample an image, we generate a set of points $P_s \subset (0,1)^2$ of predefined magnitude. These points are used to compute the (x, y) values of the feature space and pixel coordinates x_p, y_p. For sampling point $s \in P_s$, the pixel coordinates are computed as follows

$$\begin{aligned} x_p &= \mathrm{round}(s_x \cdot (\mathit{width} - 1)), \\ y_p &= \mathrm{round}(s_y \cdot (\mathit{height} - 1)), \end{aligned}$$

where *width* and *heigth* represent the size of the sampled image in pixels and the round() function is standard rounding to the nearest number from $\mathbb{N}$. The feature properties x, y are computed from pixel coordinates x_p, y_p by normalization back to $\langle 0, 1)$ range:

$$x = \lfloor x_p / width \rfloor,$$
$$y = \lfloor y_p / height \rfloor.$$

This way, the feature coordinates are snapped to the pixel borders, thus they more adequately correspond to other properties gathered from the pixel (i.e., the color and the surrounding texture).

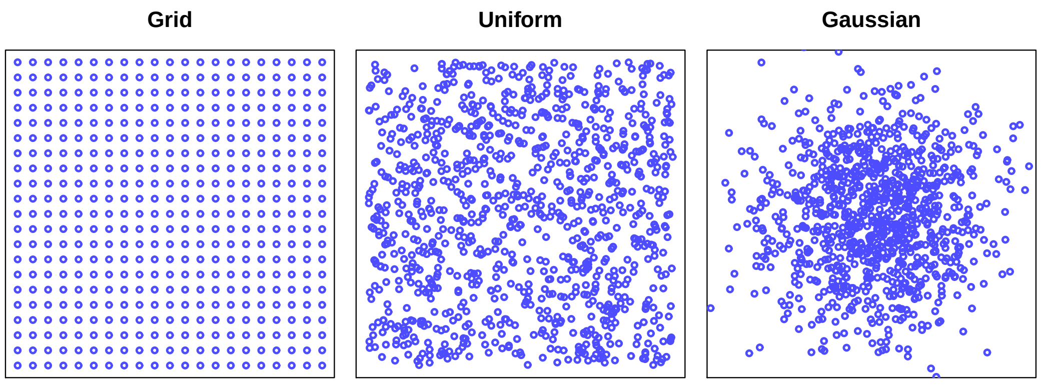


Figure 5.1: Examples of sampling distributions

The sampling set P_s can be generated several ways, some of which are depicted in Figure 5.1. We can use a regular pattern (a grid for instance) or random pattern. In case of random patterns, there are various distributions that can be used such as uniform distribution where all points of $(0, 1)^2$ are selected with the same probability, or Gaussian distribution, which tends to select more points from the center of the image. In our work, we have chosen a random selection of gaussian distribution with mean value $(0.5, 0.5)$, as it helps avoiding alias effect of regular sampling and the center of the image often contains the most important objects from the perspective of similarity.

Color Information

The color information is extracted from the pixel, at the feature coordinates. Optionally, an average color from some small surrounding can be computed as the representative color, but since we already use rather small thumbnails, the pixel represents sufficiently large area of the image. There are many color spaces that can be used to represent the information. Most of them use three-component systems, so we assume the color information is stored in three dimensions of the feature space. The problem is, that we need to find a color space where Euclidean distance (which is used as the basis for the ground distance in SQFD) reflects the color similarity perception of a human eye and brain.

The image is usually represented in the RGB color space. However, the human eye is more perceptive to changes in illumination than to changes in tone of the color. Furthermore, the red, green, and blue components of the color are perceived with different intensities. A very good results have been achieved with the CIE LAB [81] (or Lab) color space, so we use this representation in our implementation.

The Lab color space represent colors by three properties L, a, and b. The L stands for lightning component, a and b are the color-opponent properties. The representation is based on nonlinearly compressed CIE XYZ color space [135], which was designed with respect to the perception properties of human eye. To convert color from RGB to Lab, we need to convert it to the XYZ first and then compress it to Lab.

Let us have color represented by values R, G, and B in RGB space, where the values are normalized to interval $\langle 0, 1\rangle$. The conversion to XYZ is performed as follows. First, the RGB values are transformed by function f_{rgb}, which is defined as

$$f_{rgb}(t) = \begin{cases} (\frac{t+0.055}{1.055})^{2.2} & \text{for } t > 0.04045, \\ \frac{t}{12.92} & \text{otherwise.} \end{cases}$$

The transformed values are denoted $\bar{r} = f_{rgb}(R)$, $\bar{g} = f_{rgb}(G)$, and $\bar{b} = f_{rgb}(B)$. Then, the XYZ values are computed as a simple linear combination:

$$\begin{aligned} X &= 0.4124 \cdot \bar{r} + 0.3576 \cdot \bar{g} + 0.1805 \cdot \bar{b} \\ Y &= 0.2126 \cdot \bar{r} + 0.7152 \cdot \bar{g} + 0.0722 \cdot \bar{b} \\ Z &= 0.0193 \cdot \bar{r} + 0.1192 \cdot \bar{g} + 0.9505 \cdot \bar{b} \end{aligned}$$

When the color is transformed to XYZ space, we can compress it to the Lab space. The compression requires transformation f_{xyz} defined as

$$f_{xyz}(t) = \begin{cases} t^{1/3} & \text{for } t > (\frac{6}{29})^3, \\ \frac{1}{3}(\frac{29}{6}^2)t + \frac{4}{29} & \text{otherwise.} \end{cases}$$

Using this function, the L, a, b values are computed as:

$$\begin{aligned} L &= 116 f_{xyz}(Y/Y_w) - 16, \\ a &= 500(f_{xyz}(X/X_w) - f_{xyz}(Y/Y_w)), \\ b &= 200(f_{xyz}(Y/Y_w) - f_{xyz}(Z/Z_w)). \end{aligned}$$

The $X_w Y_w Z_w$ color is the reference white point used for white balancing. We use white point D65, which has values $X_w = 0.9505$, $Y_w = 1$, and $Z_w = 1.0890$.

Contrast and Entropy

The last two dimensions of the feature space are denoted *contrast* and *entropy*. They represent the texture properties in the vicinity of the sample coordinates. These features are based on changes in the illumination, thus they are computed from an image converted to greyscale. A square area around the selected pixel called the *lookup window* is established in the image. Pixels in the lookup window are scanned and *co-occurrence* matrix Γ is constructed [136]. The contrast and entropy values are then computed from this matrix.

Greyscale value (i.e., the illumination component) of a pixel is computed as weighted sum of its RGB components. The greyscale is quantized to fixed number of discrete values. We denote G_s the maximal value on the scale, and the values are zero based (i.e., the greyscale goes from 0 to G_s). The quantized greyscale value is computed from normalized RGB color as

$$G = \text{round}((0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B) \cdot G_s).$$

Once we have a greyscale representation of the image, a lookup window for the sample is established. If the x_p, y_p are the pixel coordinates of the sampled point, the top-left corner of the window will be $[\max(0, x_p - r_{ce}), \max(0, y_p - r_{ce})]$ and the bottom-right corner will be $[\min(width, x_p + r_{ce}), \min(height, y_p + r_{ce})]$, where $r_{ce} \in \mathbb{N}$ is the contrast-entropy window radius and $(width, height)$ are the proportions of the image in pixels. So the window is usually a rectangle comprising $(2r_{ce} + 1)^2$ pixels, but it is cropped if the sampled point is too close to the border of the image.

The co-occurrence matrix Γ is a rectangular matrix of $G_s \times G_s$ elements. Each element $\gamma_{i,j}$ is in fact the number of adjacent pixel pairs (horizontally, vertically, or diagonally) within the lookup window, where one of the pixels has greyscale value i and the other one j. If we denote $G_{x,y}$ the greyscale value of pixel $[x, y]$, the formal definition will be

$$\gamma_{i,j} = \sum_{x=left}^{right-1} \sum_{y=top}^{bottom-1} \Big(t_{i,j}(G_{x,y}, G_{x+1,y}) + t_{i,j}(G_{x,y}, G_{x,y+1}) + \\ + \; t_{i,j}(G_{x,y+1}, G_{x,y+1}) + t_{i,j}(G_{x+1,y}, G_{x,y+1}) \Big),$$

where *left* $= \max(0, x_p - r_{ce})$, *right* $= \min(width, x_p + r_{ce})$, *top* $= \max(0, y_p - r_{ce})$, and *bottom* $= \min(height, y_p + r_{ce})$. The testing function $t_{i,j}$ is a binary operator that returns 1 if the test succeed, and 0 when it fails:

$$t_{i,j}(g_1, g_2) = \begin{cases} 1 & \text{if } g_1 = i \wedge g_2 = j, \\ 0 & \text{otherwise.} \end{cases}$$

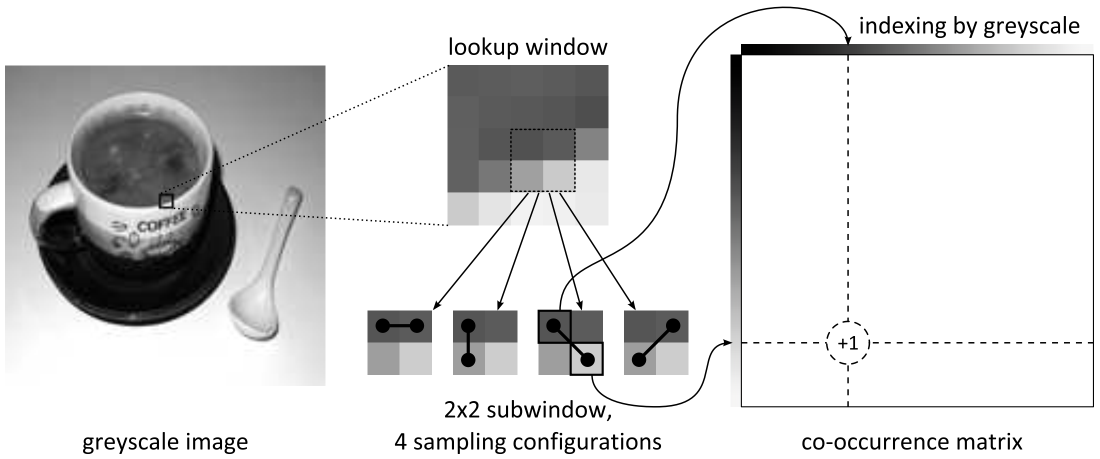


Figure 5.2: Construction of co-occurrence matrix from greyscale lookup window

The principle of co-occurrence matrix construction is depicted in Figure 5.2. To compute contrast and entropy, we require the co-occurrence matrix to be symmetric and normalized. For this reason, we define a *normalized co-occurrence matrix* $\bar{\Gamma}$ as follows:

$$\bar{\gamma}_{i,j} = \begin{cases} (\gamma_{i,j} + \gamma_{j,i})/n & \text{for } i \neq j, \\ \gamma_{i,j}/n & \text{for } i = j. \end{cases}$$

The normalization constant n is the sum of all items in the original co-occurrence matrix Γ. It also happens to be the size of the lookup window (reduced by one from each side) multiplied by four, as each 2×2 subwindow of this window contributes to the co-occurrence matrix four times by the testing function $t_{i,j}$.

$$n = \sum_{i,j} \gamma_{i,j} = 4(\mathit{right} - \mathit{left})(\mathit{bottom} - \mathit{top})$$

Finally, the contrast c and entropy e is defined as follows. Let us emphasize that only a triangular submatrix is summed as the normalized co-occurrence matrix has been symmetrized.

$$c = \sum_{i=0}^{G_s} \sum_{j=0}^{i} (i-j)^2 \cdot \bar{\gamma}_{i,j}$$
$$e = \sum_{i=0}^{G_s} \sum_{j=0}^{i} - \ln(\bar{\gamma}_{i,j}) \cdot \bar{\gamma}_{i,j}$$

5.2.3 K-means Clustering

The feature samples extract the important image properties, but the information about them is scattered. We need to perform some kind of data mining to extract important trends. An obvious choice in this situation is to perform clustering on the feature space. The clusters will represent important parts of the image with similar color and texture.

Our solution is based on K-means clustering algorithm [137], which we have modified for the needs of feature extraction. The algorithm tries to group together feature samples based on their proximity in the feature space. To determine distances in the feature space, standard L_p metric on $\mathbb{R}^7$ is used. First, we revise a standard K-means algorithm. The modifications made for the purposes of the feature extraction are described afterwards.

The K-means Algorithm

The k-means is an approximation algorithm for a cluster analysis problem that tries to partition a set of observations into k clusters, where each observation is assigned to a cluster with the nearest mean value. Therefore, the clustering creates space partitioning into Voronoi cells [138]. Since the original clustering problem is NP-hard, the k-means use an iterative refinement approach to get an approximate partitioning.

The k-means algorithm takes set S of points from $\mathbb{R}^d$ and produce a set of clusters C, where $|C| = k$, $\bigcup C_i \in C = S$, and $\forall C_i, C_j \in C : i \neq j \Rightarrow C_i \cap C_j = \emptyset$. Algorithm 5.1 presents the basic idea of the iterative refinement.

```
Data: points $S \subset \mathbb{R}^d$, number of clusters $k$ ($|S| \geq k$)
Result: set of clusters $C$, so that $|C| = k$
select $(m_1^{(1)}, \dots m_k^{(1)})$, $m_i^{(1)} \in S$                 // initial mean set
$\mathcal{I} \leftarrow 0$                                            // iteration counter
while $\mathcal{I} \leq 1 \vee C^{(\mathcal{I})} \neq C^{(\mathcal{I}-1)}$ do
    $\mathcal{I} \leftarrow \mathcal{I} + 1$
    // Assignment step:  new assignment of points to clusters
    for $i = 1, \dots, k$ do
        $C_i^{(\mathcal{I})} \leftarrow \{x \in S : \|x - m_i^{(\mathcal{I})}\| \leq \|x - m_j^{(\mathcal{I})}\|, \forall j = 1, \dots, k\}$
    end
    // Update step:  compute new means as cluster centroids
    for $i = 1, \dots, k$ do
        $m_i^{(\mathcal{I}+1)} = \frac{1}{|C_i^{(\mathcal{I})}|} \sum_{x_j \in C_i^{(\mathcal{I})}} x_j$
    end
end
$C \leftarrow C^{(\mathcal{I})}$
```

Algorithm 5.1: Standard k-means algorithm

At the beginning, an initial set of mean approximates $m_i^{(1)}$ is selected from S. There are some elaborate approaches to this step, but usually a random selection is sufficient. After that, the algorithm iteratively computes the cluster assignment using the mean values. Each cluster $C_i^{(\mathcal{I})}$, where $i = 1, \dots, k$ is the cluster index and $\mathcal{I} \in \mathbb{N}$ is the number of iteration, is constructed as a set of points, for which the $m_i^{(\mathcal{I})}$ is the closest mean value in the terms of selected L_p distance. Each point is always assigned to only one cluster, even when it lies at the border of two Voronoi cells.

After the first step, where new assignment of points to clusters is established, the mean values need to be updated. The new means are computed as centroids of their corresponding clusters. Coordinates of the centroid are computed as an average value in each dimension of all points in the cluster.

The algorithm ends when the approximation becomes stable (i.e., when two subsequent assignments $C^{(\mathcal{I})}$ and $C^{(\mathcal{I}+1)}$ are identical). The last computed assignment $C^{(\mathcal{I}_{last})}$ is then yielded as the result.

Until now, we have assumed that each iteration assigns at least one point to every cluster. However, there is a possibility, that a cluster ends with no points at all. In such case, we can create a new cluster as a replacement by dividing one of the other clusters in two.

K-means Specialization for Feature Extraction

Usually, the most interesting information gathered from the clustering is the assignment of points to clusters. However, our objectives are slightly more specific. In cooperation with domain experts from SIRET research group [139], the following major modifications were proposed to the k-means algorithm:

- Most importantly, we are performing clustering to get the cluster centroids as feature space representatives and their weights, which are computed from the numbers of points assigned to each cluster. Since we do not require the assignment information, we can avoid explicit construction of the $C_i^{(\mathcal{I})}$ sets.

- The number of centroids in the signature should reflect the complexity of the image. Hence, the k-means algorithm needs to be modified to select the number of clusters adaptively. We have chosen a pruning approach. The algorithm starts with k clusters and the cluster can be removed under some conditions.

- There are two conditions upon which the clusters are being removed. Very small clusters can be pruned as they are likely to represent some insignificant part of the image or even contain information noise. Furthermore, when two mean values are close together, one of them can be removed and the corresponding clusters can be merged.

- When clusters are being pruned, the main loop condition must be altered, as there may be no stable state in which the algorithm ends. A fixed number of iterations was chosen since it is a simple solution that works very well.

```
Data: points $S$, parameters $k$, $C_{min}$, $d_{min}$, $\mathcal{I}_{max}$
Result: centroids $M$ ($|M| \leq k$), weights $w_{m_i}$ ($\forall m_i \in M$)
$M^{(0)} \leftarrow (m_1^{(0)}, \dots m_k^{(0)})$, $m_i^{(0)} \in S, w_{m_i}^{(0)} \leftarrow 0$        // initial mean set
$\mathcal{I} \leftarrow 0$                                                // iteration counter
while $\mathcal{I} \leq \mathcal{I}_{max}$ do
    foreach $m_i^{(\mathcal{I})} \in M^{(\mathcal{I})}$ do                          // prune small clusters
        if $w_{m_i}^{(\mathcal{I})} < C_{min} \cdot \mathcal{I}$ then $M^{(\mathcal{I})} \leftarrow M^{(\mathcal{I})} \setminus \{m_i^{(\mathcal{I})}\}$  // rem. cluster $i$
    end
    foreach $m_i^{(\mathcal{I})}, m_j^{(\mathcal{I})} \in M^{(\mathcal{I})}$ do  // join clusters with close means
        if $\|m_i^{(\mathcal{I})} - m_j^{(\mathcal{I})}\| < d_{min}$ then $M^{(\mathcal{I})} \leftarrow M^{(\mathcal{I})} \setminus \{m_j^{(\mathcal{I})}\}$  // merge $i, j$
    end
    $\mathcal{I} \leftarrow \mathcal{I} + 1$                                   // going for a new iteration
    foreach $m_i^{(\mathcal{I}-1)} \in M^{(\mathcal{I}-1)}$ do
        prepare $m_i^{(\mathcal{I})} \leftarrow \vec{0}$ and $w_{m_i}^{(\mathcal{I})} \leftarrow 0$
        $M^{(\mathcal{I})} \leftarrow M^{(\mathcal{I})} \cup \{m_i^{(\mathcal{I})}\}$
    end
    foreach $x \in S$ do
        find $i$, so $\|x - m_i^{(\mathcal{I})}\| \leq \|x - m_j^{(\mathcal{I})}\|, \forall j = 1, \dots, k$
        $m_i^{(\mathcal{I})} \leftarrow m_i^{(\mathcal{I})} + x$                                // (per dimension)
        $w_{m_i}^{(\mathcal{I})} \leftarrow w_{m_i}^{(\mathcal{I})} + 1$
    end
    $\forall m_i^{(\mathcal{I})} \in M^{(\mathcal{I})} : m_i^{(\mathcal{I})} \leftarrow m_i^{(\mathcal{I})} / w_{m_i}^{(\mathcal{I})}$                // (per dimension)
end
$M \leftarrow M^{(\mathcal{I})}$ and $w_{m_i} \leftarrow w_{m_i}^{(\mathcal{I})}$
```

Algorithm 5.2: Adaptation of k-means algorithm for feature extraction

These modifications are presented in the Algorithm 5.2. Since the explicit construction of $C_i^{(\mathcal{I})}$ was omitted, the assignment and update steps were merged. Each cluster i is represented only by its mean m_i and weight w_i, which is equal to the number of points that fall into that cluster. Hence, the pruning steps are performed only on the sets of means and weights.

5.2.4 Extraction Parameters

Each part of the feature extraction process has many configuration parameters, like the number of sampling points, or the number of iterations of the modified k-means algorithm. These parameters are important, since they affect the quality of the extracted features in the means of similarity search precision and indexability of the database. One of our objectives is to find optimal parameter combination that will create the best signatures possible.

Sampling Parameters

Parameters that affect the feature sampling procedure are listed in the following table.

Name	Description
P_s	Initial set of sampling points from $(0,1)^2$.
G_s	Maximal value of the greyscale. The greyscale goes from 0 to G_s. The value 15 means, that each pixel is represented by 4 bits.
r_{ce}	Radius of the lookup window. The window has size $(2r_{ce}+1)^2$ (e.g., in case of $r_{ce} = 3$, the window is 7×7 pixels large).
f_m, f_a	Linear transformation of the feature space. Each feature is transformed after sampling as $f' = f_m \cdot f + f_a$, where $f_m, f_a \in \mathbb{R}^7$. In our similarity model, the translation of dimensions have no effect, thus we use $f_a = (0,0,0,0,0,0,0)$ in all settings.

The initial point set P_s consist of several thousand points generated randomly with normal (Gaussian) distribution. The G_s limit is usually selected as $2^i - 1$, so the grey scale fully utilizes i bits. The 16-value scale was empirically observed as a good compromise. The r_{ce} value is usually smaller or equal 5 when image thumbnails of 150×150 pixels are used. Larger lookup windows overlap greatly, thus producing less localized information for the feature samples.

Clustering Parameters

The parameters that affect k-means clustering are listed below.

Name	Description
k	The number of initial seeds (and the maximal number of clusters).
$\mathcal{I}_{max}$	The number of refining iterations.
C_{min}	Constant used for pruning small clusters. In each iteration, clusters smaller than $C_{min} \cdot \mathcal{I}$ are removed.
d_{min}	Joining distance for mean values. If the distance of two means is smaller than this parameter, one of the cluster is dissolved.
L_p	A function used to measure distance between mean values. Usually, L_2 (Euclidean) distance is used.

The number of initial seeds k is better to keep higher as the clusters get pruned quite intensively. If the signature of an average image should end with 50-100 centroids, there must be at least several hundred of initial seeds.

All these parameters affect precision of the similarity model and indexability of the database, which corresponds closely with efficiency. We have conducted extensive experiments to find optimal configuration. Some of these experiments and their results are presented in Section 5.4.3.

5.3 GPU Implementation

This section presents our contributions to the field. We propose an implementation of the feature extraction process that utilizes computational power of modern GPUs. Before describing our proposed solution, a brief analysis of the problem is performed in Section 5.3.1. It is followed by the overall description of the extractor architecture and two the most important algorithmic parts – feature sampling process and k-means clustering.

5.3.1 Problem Analysis

The extractor is designed to process many images in single batch, as it is usually used to index the entire database. It can be used to extract features from one or only a few images, but since it is rarely the case and the response time has much lower priority than processing throughput, we do not optimize for this special case.

The situation is similar to block-wise distance computations from the previous chapter. The entire collection of images is unlikely to fit the GPU(s), thus an iterative approach must be used. For this reason, we utilize the GPU framework presented in Section 3.4, which was successfully employed for the distance computations.

The iterative approach computes signatures in batches called blocks. The question remains, how the workload is mapped to GPU threads and work groups. The most direct solution would be to assign one thread per image (signature) in the block. However, this solution is unfeasible for the same reasons it was not employed for SQFD computations. Feature extraction of one image is quite complex task that requires rather large portion of local memory to cache the data and intermediate results. Furthermore, images of different complexities produce imbalanced workload for the threads running in lockstep and scattered memory access patterns. Many steps of the extraction process can be performed concurrently, so we can use more fine grained parallelism. Communication and synchronization between work groups is always complicated, so we assign one work group to extract signature from one image. The threads in the work group cooperate to perform each step in parallel manner.

Data

The block of input images is best to store in one continuous array as it makes the upload from host memory much faster. If all images are normalized to the same size, the offset of each image can be easily computed. Otherwise, an index has to be built, so each work group can locate its image and determine its size. The situation with output data is a little bit more complicated. The signatures differ in their sizes, thus the lengths have to be saved along with the signatures. Furthermore, we need to deal with fact, that the global memory for the results has to be allocated before the kernel starts.

The extraction process itself has two phases. The first phase reads the input image and produces a set of feature samples. The feature samples are used as input for modified k-means clustering. It produces the centroids and weights, which in fact form the feature signature. The input images and the output signatures has to be stored in global memory due to data transfer limitations from/to host memory. The sampled features are accessed only by threads in one work group, so it would be best to keep them in the local memory of the SMP. Unfortunately, the local memory of current GPU architectures cannot accomodate the number of samples we require, thus they need to be stored in global memory as well.

The computation of contrast and entropy in the sampling process requires the image data converted to greyscale. We can convert the colors on the fly or we can create an explicit copy of the image in greyscale representation. Greyscale value of many pixels are required multiple times as the lookup windows of some samples may overlap and each lookup window is scanned with 2×2 subwindow. It is rather difficult to cache converted colors, so a greyscale copy of the image seems to be a better choice, especially if we manage to keep it in the local memory.

The co-occurrence matrix, which is also required for computation of contrast and entropy, is small enough to fit the local memory for all reasonable values of G_s. Actually, the local memory can accomodate multiple matrices, so we can compute multiple samples concurrently.

The subsequent clustering phase requires the mean values and the weights that represent the clusters. In fact, we need two subsequent copies of the means and weights as the new copy is computed the from the last copy in each iteration. These values are accessed multiple times when nearest mean is being found for each point. Furthermore, they are accessed in a highly random pattern when the new copy is computed. For these reasons, we chose to limit the maximal number of clusters, so the means and weights can be kept in the local memory during the clustering.

Profiling and Code Analysis

For the purposes of optimization, we have profiled the code of the extractor to determine, which steps take the most time and will benefit from optimizations the most. We do not present any exact results as they vary significantly based on configuration parameters, but only a general overview.

The most expensive of the feature sampling phase is the computation of contrast and entropy. The explicit construction and processing of co-occurrence matrix for each sample requires much more time than a straightforward conversion formula that computes Lab color from RGB.

In the k-means clustering, the most time is spent by finding the assignment of each point to its nearest cluster. This step takes $\mathcal{O}(nk)$ time, where n is the number of points and k is the number of clusters, as the distance between every point and every cluster mean has to be computed and tested. There are specific data structures (like kd-trees [140]), which can accelerate the search process; however, they are difficult to implement efficiently in the GPU memory and they would not help much, since we have limited the number of clusters significantly.

The pruning steps require $\mathcal{O}(k)$ and $\mathcal{O}(k^2)$ time, so they are much cheaper as k is significantly smaller than n. The cluster joining step is more complicated than filtering of small clusters as it requires either serial execution or a fine grained synchronization. We address this problem later, in Section 5.3.4.

5.3.2 The Design of the Extractor

The extractor is built using OpenCL library and the framework presented in Section 3.4. This section presents the high-level design of the extractor and the format of global memory data structures.

The Architecture

External architecture of the extractor is very similar to the architecture of GPU accelerated SQFD engine, that does not employ any metric access method (i.e., the sequential search algorithm). The images to be extracted are grouped to blocks, and dispatched via feeding threads to the GPUs. Each GPU has attached two feeding threads and there are two blocks dispatched to each GPU (one block is being computed, one block is being transferred). The feeding threads are responsible for gathering all images into single memory block, so they can be copied to GPU device in one transaction. The signatures received from GPU are scattered due to different sizes of the signatures. They are compacted by feeding threads postprocessing.

The internal architecture of the extractor is depicted in Figure 5.3. As mentioned before, each image is processed by one work group. Sampled features, which are produced by the first phase of the extraction, are kept in global memory. Even though there is only one buffer required per running work group, we statically allocate one buffer per image, so the work groups do not have to claim/release the buffers dynamically. The entire local memory is allocated for the work group as one large buffer and the threads use different partitioning of the buffer for different steps of the extraction algorithm.

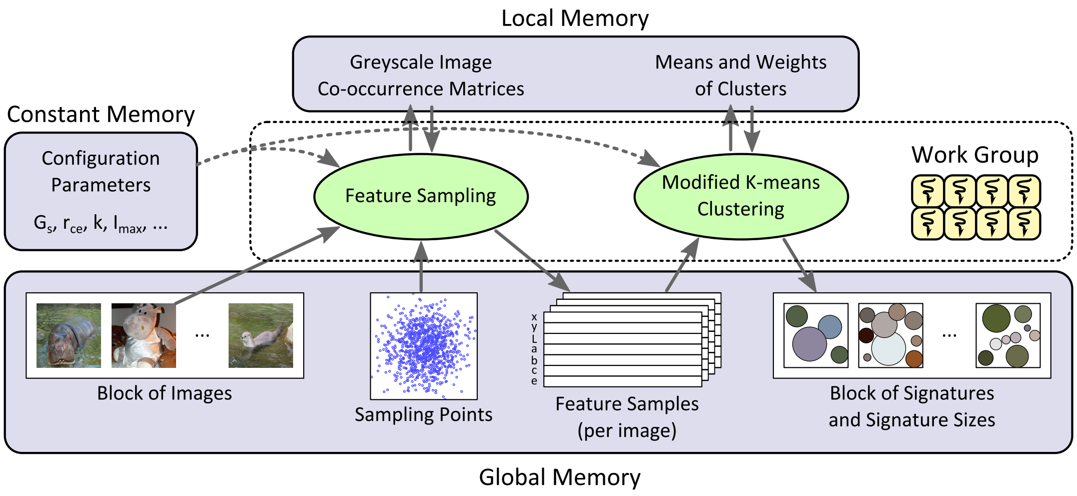

Local Memory
Greyscale Image Co-occurrence Matrices
Means and Weights of Clusters
Constant Memory
Configuration Parameters
G_s, r_ce, k, I_max, ...
Feature Sampling
Modified K-means Clustering
Work Group
Block of Images
Sampling Points
x y L a b c e
Feature Samples (per image)
Block of Signatures and Signature Sizes
Global Memory

Figure 5.3: Internal architecture of the extractor

Data Representation

The images are stored in one continuous array, each image linearized in row-wise manner. The color values of the pixels are stored in RGB format, where each channel is encoded in 8 bits, aligned to 32-bit words, so each pixel can be loaded by one read operation. All the images have the same size (150×150 in our case) and the resolution $(width, height)$ is kept along with the configuration parameters.

The feature signatures are represented in the same column-oriented format, which is used for distance computations as described at the end of Section 4.4.2. The only problem is, that individual signatures differ in size. For this reason, we allocate memory buffer for the signatures as if each signature has maximal possible number of clusters k. The starting offset of each signature is aligned to the maximal size of the signature, so it can be easily computed. However, the signature may not use its entire preallocated space.

A second buffer is allocated for the signatures, where each work group writes the size of the signature it has extracted. These sizes are used to compact the signature buffer and to create index for the signatures. The compacting and indexing is performed by the feeding threads after the data are transferred to the host memory.

The intermediate buffers for sampled features use the same column-wise representation as the signatures. Each buffer consist of seven arrays, each one containing $|P_S|$ 32-bit float numbers. Adjacent threads in the group always work with adjacent sampled features, so the loading and writing of the feature values is executed by the whole warp at once, thus in coalesced manner.

The configuration parameters are described in Section 5.2.4. All of them, except for the set of sampling points P_S and chosen L_p distance function are stored in a data structure in constant memory. All the values are either real numbers or integers, so they can all be stored in 32-bit words. The sampling points P_S are stored in global memory in column representation (i.e., as array of x values followed by array of y values) and the size of the set $|P_S|$ is kept with the configuration parameters. The L_p is hardwired in the kernel code, but it can still be selected in the runtime as the kernel compilation is performed dynamically by OpenCL.

5.3.3 Sampling Features

As described before, the first phase of the extraction process is the feature sampling. The process takes image data in RGB format and the set of initial points to produce a list of feature samples. Both input and output data are stored in the global memory of the GPU as neither fit the local memory and the local memory is utilized to store the intermediate data.

The sampling is performed in three steps. First, the color information and the spatial coordinates are extracted. Then, the entire RGB image is converted to its greyscale representation, which is stored in local memory. Finally, the greyscale image is used to compute contrast and entropy values for each sample. We describe these steps in more detail.

Color Information

The color extraction is quite straightforward. Each sample can be extracted independently and there are more than enough samples to occupy all the threads. Equations for converting color from RGB space to CIE LAB space have been described in Section 5.2.2. These equations are computed by fixed number of instruction, thus they provide a stable workload for the threads.

The color information of each pixel is encoded in 32-bit word, so each thread loads exactly one value. Threads are accessing the pixels randomly as they use randomly generated sampling points from P_s. Unfortunately, 150×150 image with 32-bit encoding requires $90,000$ bytes of space, so it cannot be cached in the local memory of current GPUs. On the other hand, the situation is not that serious as each pixel is required only once in this step and the inefficiency of data access pattern is moderated by L1 and L2 caches.

The following step requires also the color information of the pixels. We have considered several possibilities, how to cache at least some of the pixels loaded from global memory, but we were unable to design any mechanism that would improve the performance.

Greyscale Image

The computation of contrast and entropy requires a lookup window for each sample. Theoretically, we could load and convert only those parts of the image, but in most cases, these windows cover significant amount of the image pixels[1]. In this case, it is better to convert entire image into greyscale bitmap even though some of the pixels are converted in vain. The unnecessary work performed is significantly outweighed by the improvement of performance, as the conversion of the entire image produces much more regular workload and data access pattern.

The greyscale bitmap is encoded in highly packed format, which respects the 32-bit word boundaries as the GPU works natively with 32-bit values. Each word encodes $\lfloor 32/\lceil log_2(G_s+1)\rceil \rfloor$ pixels, since $\lceil log_2(G_s+1)\rceil$ is the smallest amount of bits required for encoding values from 0 to G_s.

[1] For instance, if we use two thousand samples with 7×7 lookup window, total $98,000$ pixels are traversed. But 150×150 image has only $22,500$ pixels, thus it would take less work to convert the entire image.

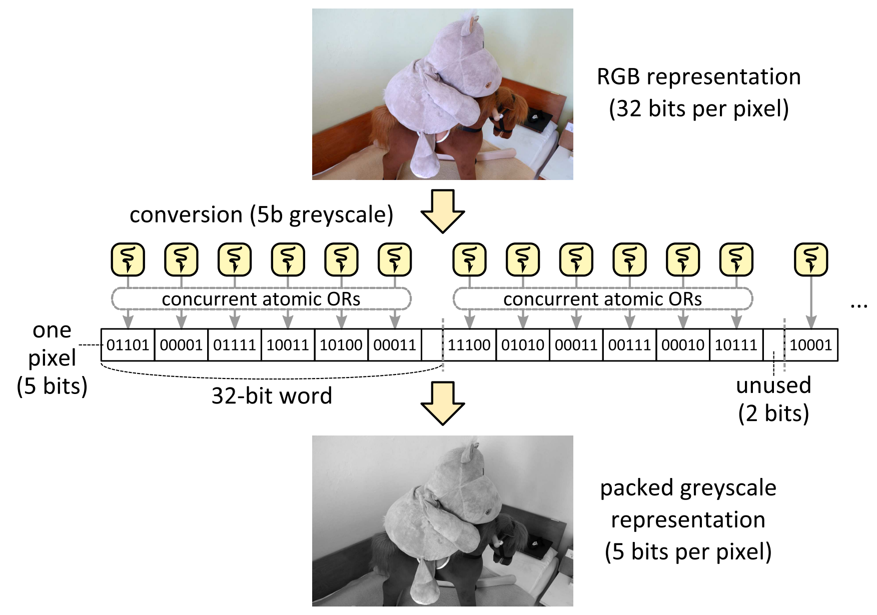


Figure 5.4: Construction of 5-bit packed greyscale image in SIMT mode

The construction process is illustrated in Figure 5.4. The space allocated for the bitmap in the local memory is first filled with zeros. After that, the threads process independent pixels, compute the greyscale value from the RGB color and use atomic OR operation to save the value into the bitmap. The threads need to use logical OR as each numeric operation is performed on 32-bit words and the pixel is represented by only a few bits. The OR must be atomic as multiple threads may write to the same word at the same time.

This method clearly causes bank conflicts as adjacent threads often write not only to the same memory bank, but to the same memory cell. On the other hand, loading data from global memory takes much longer than writing to local memory, so the loads of one warp will overlap with stores of another one even though the writes are slowed down several times by serialization.

Contrast and Entropy

For each contrast and entropy value a co-occurrence matrix must be computed first. This matrix must be explicitly constructed, so a memory buffer has to be allocated for it. Even though each lookup window could be traversed concurrently by multiple threads, the final contrast and entropy values are better to be computed serially. Furthermore, the lookup window is usually quite small to truly benefit from parallel processing, especially when the synchronization of increments in co-occurrence matrix has to be considered.

For these reasons, we have decided to use the same approach as for color extraction. Each thread in the group computes contrast and entropy values (thus the co-occurrence matrix) for a different feature sample. This means that multiple co-occurrence matrices have to be stored in the local memory. The co-occurrence matrix uses the same compact format as the greyscale bitmap. Each 32-bit word encode $\lfloor 32/\lceil \log_2(4(2r_{ce})^2)\rceil \rfloor$, since $(2r_{ce}+1)^2$ is the size of the lookup window and each 2×2 sub-window produces four increments in the co-occurrence matrix.

For instance, when 4-bit greyscale is used ($G_s = 15$) and $r_{ce} = 3$ (the lookup window of 7×7 pixels), the local memory is utilized as follows. The greyscale bitmap stores 32/4 pixels in each word (two in each byte), so the 150×150 image takes $11,250$ bytes. One co-occurrence matrix has $(G_s + 1)^2 = 256$ elements and $4(2r_{ce})^2 = 144$, so each element fits in one byte. Therefore, we can store $\lfloor (49,152 - 11,250)/256 \rfloor = 148$ matrices in the local memory along with the greyscale bitmap.

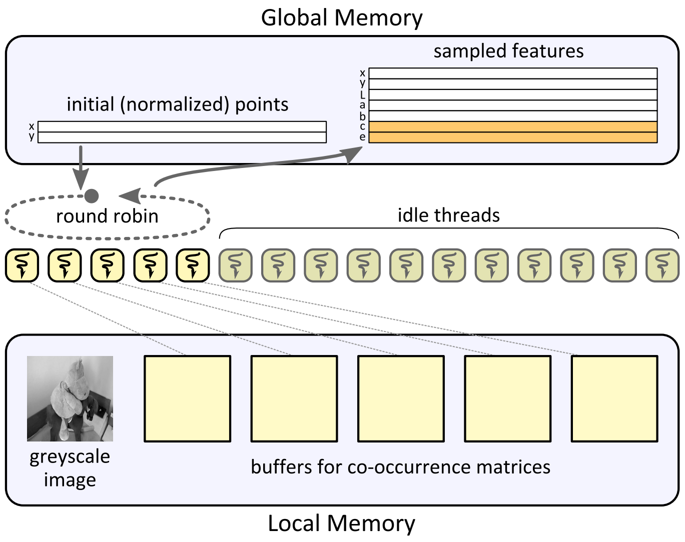


Figure 5.5: Computing contrast and entropy by the entire work group

The algorithm calculates, how many co-occurrence matrices can be allocated in the local memory based on parameters G_s, r_{ce}, and the size of the memory. A thread from the work group is assigned to each matrix and threads without the matrix become idle. Active threads load the sampling points in round robin fashion, construct their own co-occurrence matrix, and compute corresponding contrast and entropy. The schema of the algorithm is depicted in Figure 5.5.

The construction of each co-occurrence matrix still uses logical bit operations to update the values in compacted format; however, these operations need not to be atomic as each matrix is constructed by just one thread. The bank conflicts during matrix construction cannot be avoided since the access pattern is completely dependent on the image data. To avoid bank conflicts when the matrices are read, their size can be aligned to the nearest greater number which is not divisible by the number of banks.

In case the number of allocated matrices is higher than the size of the warp, it might be beneficial to reduce their amount. If the working threads and the idle threads are aligned to the warp boundary, all the threads in each warp either compute or skip the computation, therefore the SMP cores will be utilized better.

5.3.4 K-means Clustering

It has been already established that the sampled features cannot be cached in the local memory of the SMP. However, we can store up to 1536 mean values and weights in 48 kB of memory. We do not need to keep means and weights from each iteration. No more than two subsequent iterations have to coexist in the memory at any given time as one version is used to compute the next. Therefore, there can be up to 768 seeds for the k-means on current GPU architectures. This amount was empirically verified to be more than sufficient.

The SIMT code uses similar outline as Algorithm 5.2. All threads in the work group advances through the main loop together and there is a barrier after each step. The individual steps are preformed in SIMT manner whenever possible.

At the beginning, the the threads cooperatively copy the first k feature samples as initial mean values. The sampling points are randomized, so we can take any part of them as a random subset. The weights are initialized to zeros. After that, $\mathcal{I}_{max}$ iterations of the main loop are performed. Last version of the mean values and their corresponding weights is written cooperatively by all threads in the group to the designated arrays in global memory. The number of clusters that remained after the last iteration is written to the global index by the leading thread of the group.

Computing Next Version of Means

The next version of the mean values and their weights is computed in three steps. An explicit synchronization on a barrier is performed after each one of them.

1. The buffers for the next version of means and weights are filled with zeros.
2. The sampled features are scanned in parallel and the nearest cluster is found for each one. The feature value is added (per dimension) to the new mean of the nearest cluster and its weight is incremented.
3. Coordinates of the new means of nonempty clusters are divided by their corresponding weights (to compute a centroid of the newly formed cluster).

All three steps are conducted in parallel by all threads in the group. The workload is processed by the threads in round robin fashion. The only problem is that two threads may need to add a value to the mean or increment the weight of the same cluster at the same time.

The atomic increment is available in the arsenal of the OpenCL builtin functions [38]. Unfortunately, atomic versions of arithmetic operations are restricted to integers only. The CUDA framework implements also the atomic add function for 32-bit floats, but such solution is not portable.

We have implemented emulation of atomic add using atomic compare-and-exchange function. The code is presented in Algorithm 5.3.

```
**Data**: variable $x$, number to add $a$
**Result**: update variable $x$, and return the value of $x$ before addition
$old \leftarrow x$
repeat
    $assumed \leftarrow old$
    $old \leftarrow \text{cmpxchg}(x, assumed, assumed + a)$
until $assumed \neq old$
return $old$
```

Algorithm 5.3: Emulation of atomic addition using compare-and-exchange

The `cmpxchg` function takes three arguments (var, old, new). It atomically tests, whether $var = old$ and if it does, the $var \leftarrow new$. The function returns old value if the swap was performed or the actual value of var if no swapping was done.

This emulation works quite efficiently when the collisions are rare. However, when there are only a few clusters left, it might be better to use some other approach, like variable privatization for instance. More extensive research on this particular topic is still required.

Parallel Pruning

There are two pruning steps. The first one removes all clusters that are smaller than $C_{min} \cdot \mathcal{I}$, where C_{min} is a configuration parameter and $\mathcal{I}$ is the number of current iteration. The second one tests each pair of clusters and join those which have centroids closer than d_{min}.

In the serial algorithm, the array of means and weights can be easily kept compacted. When a cluster (i.e., the mean and weight values) is being removed, its values are overwritten by the values of the last item in the array and the size of the array is decremented. Such operation has $\mathcal{O}(1)$ time complexity, so it does not affect the efficiency of the pruning. However, we cannot use the same technique in parallel execution without explicit locking of the items in the array. Instead, we only mark deleted clusters by setting their weights to zero. The array is compacted by a parallel algorithm after the pruning concludes.

At this point, we have slightly modified the original algorithm by changing the order of the pruning steps. First, the clusters with centroids closer than d_{min} are joined. Then, the small clusters are removed. Furthermore, we have merged the second pruning step with the compacting step. The compacting is designed to remove empty clusters and clusters marked for deletion. It can be easily modified to remove not only empty clusters, but clusters smaller than given constant.

The join of two clusters is in fact achieved by deleting one of them. When conducted in parallel, it may happen that two threads finds the same pair of clusters, but one of the threads decides to dissolve the first cluster and the other thread dissolves the second cluster. We could add some deterministic rules, like when clusters i, j are being joined, the cluster with greater index is removed.

Unfortunately, these rules cannot cover all the possibilities as we demonstrate on a simple example. Let us have three clusters $i < j < k$. One thread discovers that i, j are close enough to be joined and second thread wants to join j, k. If we apply rules like the one described in the previous paragraph, the first thread removes cluster j and the second one cluster k. But if the conditions were tested serially, the cluster j would be removed first, so there will be no test performed between j, k.

There are many possible solutions but most of them are quite complex. We have settled for a simple yet sufficiently efficient one. The remaining nonempty clusters are tested in steps. In each step i, the cluster i is being tested, whether it should be removed. The cluster is removed, if there is a cluster j $(j > i)$, so that means of i and j are closer than d_{min}. These steps are executed sequentially with an explicit barrier at the end. The search for cluster j is performed concurrently by all available threads. When a thread finds cluster that is close enough to i, it assigns zero to w_i, so it marks cluster i as deleted. Multiple threads may perform this assignment simultaneously but that does not change the result as it is an idempotent operation.

Our approach may be suboptimal as the synchronization after each step limits the parallelism. However, this approach is easy to implement and it produces stable results. We tried some improvements but with no measurable impact on the performance. We believe that it does not worth optimizing this step any further as the major part of the work is performed in the computations of the next version of mean values and weights.

Compacting Step

The compacting step takes one instance of mean values and weights and produce another instance, where all clusters with weights smaller than $C_{min} \cdot \mathcal{I}$ are dismissed. The remaining records shift towards lower indices to fill in the gaps, so the newly created arrays of means and weights are compact. This step is very important, since it ensures that the cluster records are always kept in the smallest array possible, thus it takes the least amount of time to scan them, when the nearest mean needs to be found for each point. The compacting step also takes care of one of the cluster pruning steps.

The compacting step has two phases. First, we compute compacting index $O[i]$ for each item i in the new array and the number of the remaining items. Index O holds the moving instructions for the following phase as the i-th mean and weight values in the new index are copied from the position $O[i]$ of the old arrays. After computing the index O, threads in the work group process items in the new arrays in round robin fashion. They copy the mean values $m'_i \leftarrow m_{O[i]}$ and the weight values $w'_i \leftarrow w_{O[i]}$, where m', w' denote new arrays and m, w denote old arrays of means and weights respectively.

The copying phase is quite straightforward, optimal in the number of operations performed, and completely data parallel. We need only to find an efficient way how to compute index O. The simplest solution would be to compute the index by one-pass scan performed by a single thread. Such approach would be easy to implement and optimal in the number of operations; however, it is strictly serial. Despite the serial nature of this approach, it can be used in the implementation since there are only several hundred clusters (due to limitations imposed by local memory size) and only a trivial test is performed on the weight value of each cluster.

We attempted to improve the solution and utilize the computational power of all cores of the SMP by employing an approach based on parallel reduction trees. First, a prefix-sum array S is constructed, so that $S[i] = |\{w_j : w_j \geq C_{min} \cdot \mathcal{I}, j = 1, \dots, i\}|$. In other words, $S[i]$ is the number of items from the range $1, \dots, i$ which are not being removed by the compacting step.

The computation of S is illustrated in Figure 5.6. At the beginning, each item $S[i]$ is set to 1 if the item i is kept, or to 0 if item i is being removed by the compacting step. The first $\log_2(|S|)$ steps corresponds to standard reduction tree that uses numeric addition as the joining operator. The following $\log_2(|S|) - 1$ steps use reverse reduction to update the remaining values in S.

When S is computed, the O is constructed as inverse index of S. The threads scan the weights again and for each $w_i \geq C_{min} \cdot \mathcal{I}$ the $O[S[i]] \leftarrow i$ is set. We can make an observation that for any two w_i, w_j, which are both greater or equal

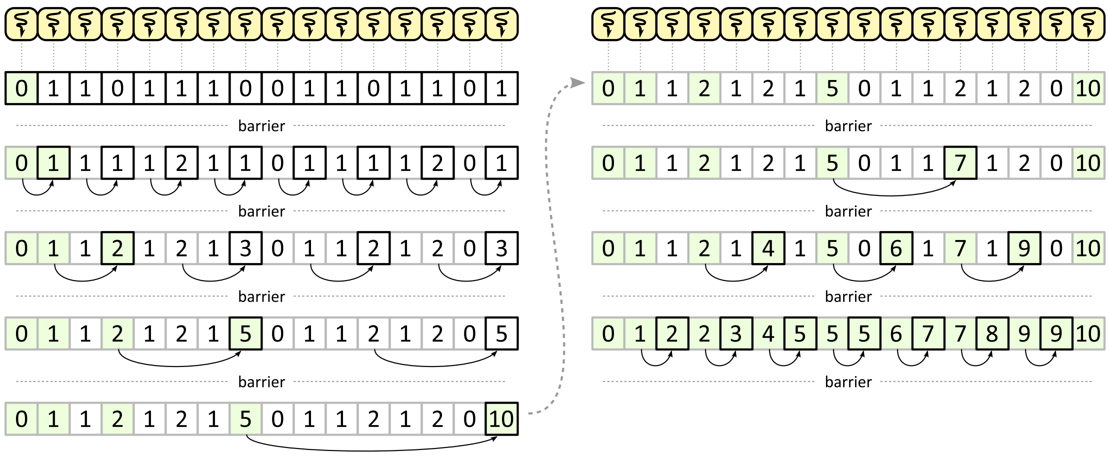


Figure 5.6: Computing prefix-sum S for 16 items

to the minimal cluster size, the $S[i] \neq S[j]$, so the writes to the index O do not collide.

Despite our effort, the overall performance improved only slightly. The dominant portion of the work is performed in the combined step that determines the assignment of each point and compute new version of mean values. However, we have tested that this approach is vital and can be used in other parallel algorithms.

5.4 Experiments

The experiments are divided into two parts. The first set of experiments was designed to test the performance of the new extractor and compare the speedup with respect to the CPU platform. The second part of this section presents some of the experiments performed to find an optimal configuration parameters for the extractor. These tests are very extensive and our primary objective was to accelerate the extraction process, so these results are presented only as an illustration of the possibilities that were opened by the GPU extraction.

5.4.1 Hardware and Methodology

The experimental hardware has already been presented in Section 4.7.1. Let us quickly revise that the GPU server is equiped with Xeon E5645 processor comprising 6 physical (12 logical) cores running at 2.4 GHz, 96 GB of DDR3-1333 RAM, and 4 NVIDIA Tesla M2090 GPU cards based on Fermi architecture. Each GPU chip have of 512 cores (32 cores per 16 SMPs) and 6 GB of memory.

Again, we tested the GPU implementation on a commodity PC with two gaming cards NVIDIA GTX 580 (512 cores and 1.5 GB of memory each). The extractor has similar performance on the gaming GPUs as on Teslas, thus we do not provide any further details.

The CPU tests were conducted only on the GPU server. The CPU implementation of the extractor scales almost lineary with the number of CPU cores, thus it is reasonable to assume, that the NUMA server with four Xeon CPUs (24 physical cores) will have approximately four times higher throughput than GPU server with single Xeon (6 physical cores).

Data

The experiments were performed on two datasets. The first one is the *Thematic Web Images Collection* (TWIC) [141], which comprises 11, 555 images. This collection is rather small for extensive performance testing, but the dataset is annotated, so we have the ground truth for precision experiments. The images are divided into 200 classes and one testing query is selected from each class.

For the performance tests we selected the Profimedia [120] dataset. Let us revise that Profimedia is a commercial image database available on the internet. Our sample consist of 17.5 million randomly chosen images. Since the entire database does not fit in RAM, we have performed two types of tests. We have selected a subset of 1 million images which can be cached in RAM to test the peak performance of the GPU extractor. We have also used the entire dataset to create more realistic experiments that incorporate the loading times of the images from the disk.

All the images in both datasets were converted to RGB bitmaps and resampled to 150×150 by bicubic interpolation. Theoretically, the image decoding and resampling can be also performed by the extractor on the GPU, but these operations are beyond the scope of this work. We have created binary files that aggregate multiple images, to reduce the loading time from the disk. The entire TWIC dataset is stored in one large file and the Profimedia was divided into blocks of 10, 000 images per file. These binary files allows us to minimize the overhead of the operating system calls, such as opening/closing the files or traversing large directories.

Methodology

The performance tests were repeated at leas three times and the time of extraction was measured using the system real time clock. If any of the measured values deviated from the average by more than 15%, the value was discarded as tainted and the test was executed again. Experiments that tested the peak performance of the extractor have the images preloaded in the operating memory. The large scale test which also measure the loading time have purged the disk I/O buffers, so no data were cached in RAM.

The precision tests that search for an optimal configuration were performed only once since they are deterministic. We have verified some of their results to ensure that the GPU extractor produces the same signatures as its CPU counterpart.

5.4.2 Performance Evaluation

The first set of tests is designed to evaluate the performance of the GPU extractor and compare it to the CPU implementation. The main portion of the tests were performed on the TWIC dataset, since we want to use them in the same perspective as the precision tests presented in Section 5.4.3. The large scale tests on the Profimedia database, which determine the peak performance of the extractor, are presented at the end of this section.

Preliminary Tests on TWIC

These tests does not reveal the full potential of the extractor, since the TWIC dataset is rather small. However, they are quite interesting as they indicate, how much the training process of the configuration parameters can benefit from the GPU acceleration. The following tests use $2,000$ points for feature sampling, where the contrast and entropy are computed using 7×7 lookup window ($r_{ce} = 3$) and greyscale of 16 values (4 bits per pixel). The k-means clustering was performed in 5 iterations, using 400 initial seeds, with parameters $C_{min} = 2$ and $d_{max} = 0.2$ while L_2 was the clustering metric.

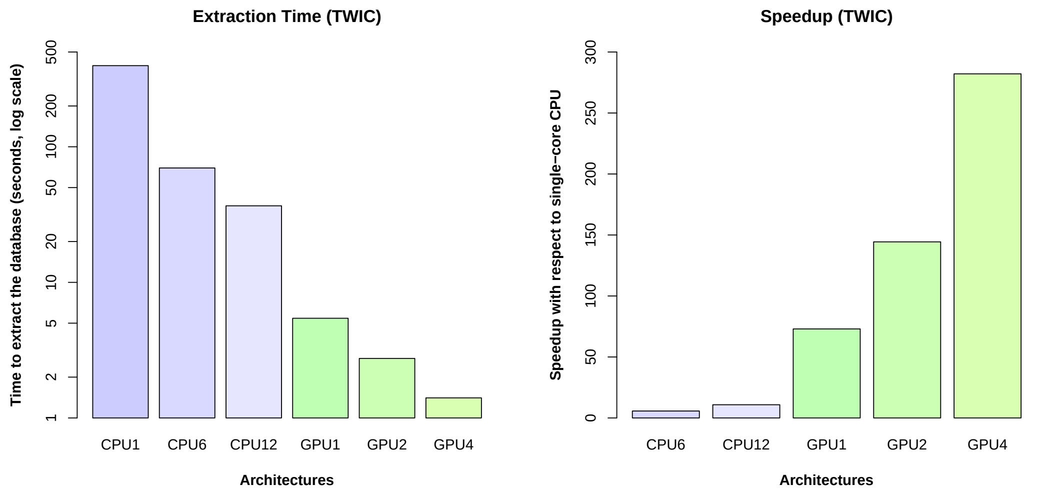


Figure 5.7: Results of preliminary performance tests on TWIC dataset

The results are depicted in Figure 5.7. The problem scales almost linearly with the number of CPU cores and GPU devices. The extraction of entire database takes approximately 400 seconds on single CPU core and 70 seconds on the 12-core CPU, but it can be done in 1.4 seconds on four GPUs. This corresponds to the $282\times$ speedup over a single-core and $50\times$ speedup over 12 CPU cores. The amount of work we can compute in one day on the GPUs would take nearly two month on a multi-core Xeon and almost a year on a single-core CPU.

Impact of Various Parameters on Performance

Some of the configuration parameters can affect the performance of the extraction. The performance of feature sampling is affected by the size of the initial set of point $|P_S|$, the grey scale size G_S, and the radius of lookup window r_{ce}. The linear transformation of the feature space does not affect the speed of the sampling process; however, it may have indirect impact on the speed of the clustering process.

The impact of the size of the sampling points set $|P_S|$ is rather obvious as the sampling process has time complexity $\mathcal{O}(|P_S|)$. On the other hand, the parameters that configure the contrast and entropy computations affect the algorithm in a more complex way, thus we have measured their impact empirically. The results are depicted in Figure 5.8. The greyscale range tests use fixed radius $r_{ce} = 3$ and the lookup window tests use fixed greyscale of 4-bits ($G_S = 15$).

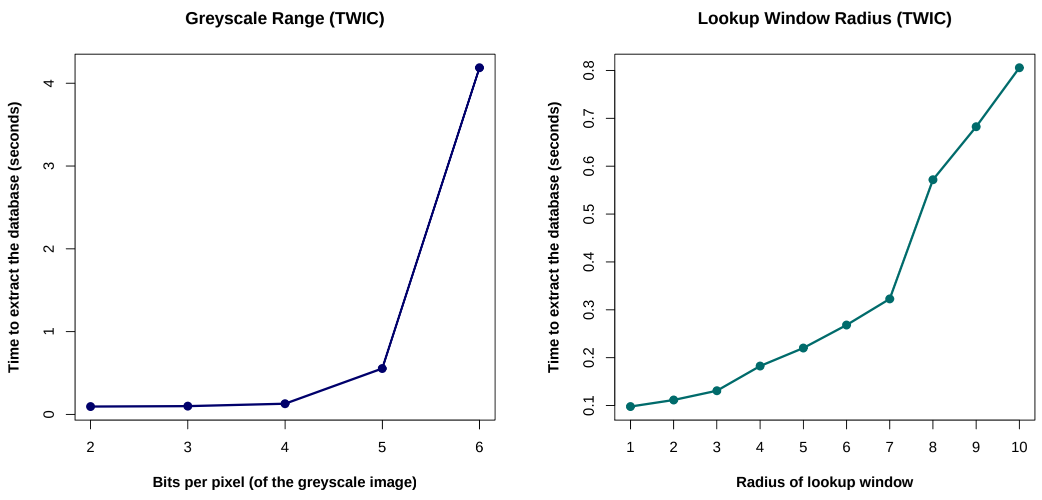


Figure 5.8: Performance impact of greyscale size (G_S) and radius of the lookup window r_{ce} for the contrast and entropy computations

The greyscale size G_S exhibit a significant drop in performance when more than 4 bits are used to represent each pixel. The reason is that the G_S parameter affects both the memory taken by the greyscale image and the size of the co-occurrence matrices used to compute contrast and entropy. When 5-bit greyscale is used ($G_S = 2^5 - 1 = 31$), the converted image takes $15,000$ B of local memory, so we can fit only 35 buffers for co-occurrence matrices of 31×31 items. This is barely enough to keep one warp of threads occupied, thus the parallelism is severely reduced. The situation gets much worse in case of 6-bit greyscale ($G_S = 2^6 - 1 = 63$). The image takes $18,000$ B of local memory, so only 7 matrices (63×63) can be stored along with the image.

We can improve the situation by two possible modifications. The co-occurrence matrix becomes quite sparse when the G_S grows and the lookup window keeps its size. A sparse representation of the matrices can be implemented, so more of them can be stored simultaneously in the memory. Second possibility is to adjust the workload mapping and let each co-occurrence matrix to be constructed and traversed by more than one thread. However, we found that both modifications might be unnecessary, as 4-bit greyscale was proven to have enough precision for the similarity model.

The size of the lookup window is quadratically dependent on the radius as it has $(2r_{ce} + 1)^2$ pixels. Furthermore, the memory required by the co-occurrence matrix depends on r_{ce}, as the range of their values are limited by $4 \cdot (2r_{ce} + 1)^2$. The significant drop in performance between r_{ce} values 7 and 8 can be explained by two causes. First of all, the co-occurrence matrix can no longer fit three values to each 32-bit word and when it switches to two values per word, the memory required by the matrix is increased by factor $3/2$. Second, we believe that at this point the workload of traversing the lookup window exceeds the workload of traversing the co-occurrence matrix and since the lookup windows are picked up by random sampling points, the number of bank conflicts on the local memory rise significantly.

The impact of two the most important parameters of the k-means clustering is depicted in Figure 5.9. We have focused on the number of iterations and the selected L_p metric. The impact of the number of initial seeds (parameter k) is rather straightforward, so we need not test it. The impact of other parameters, such as the minimal cluster size C_{min} or the joining distance d_{min}, corelates strongly with the distribution of feature samples, hence the result vary from image to image.

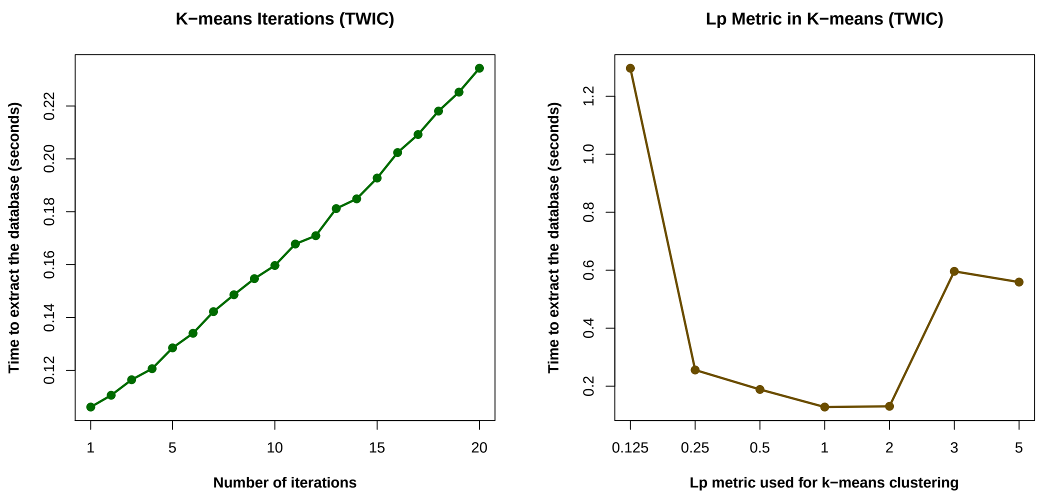


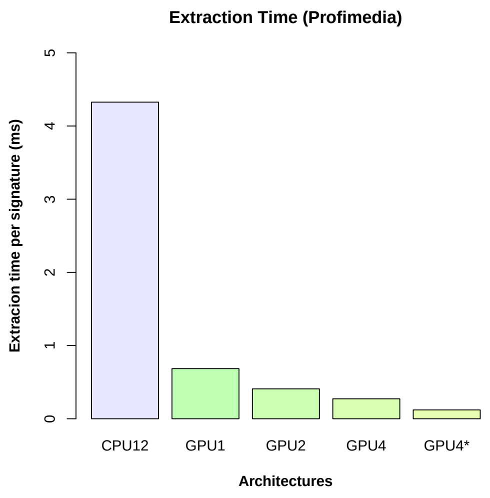
Figure 5.9: Performance impact of the number of k-means iterations and used L_p metric

As expected, the speed of the extractor is linearly dependent on the number of k-means iterations. The selected L_p metric affects the performance significantly as it is the dominant operation performed when the assignment of the points is computed. Traditional metrics like L_1 (Manhattan metric) and L_2 (Euclidean metric) are easy to compute with only a few arithmetic operations. Other metrics require significantly more computations, or even the utilization of mathematical functions, which are performed by special units.

Large Scale Tests

To verify that our extractor can sustain the same level of throughput continuously, we performed a large scale tests on Profimedia dataset. The results are summarized in Figure 5.10. The first graph presents the extraction times (in milliseconds) per signature for various architectures. The second graph shows the throughput of the extractor in signatures per second.

Results denoted CPU1 and CPU12 were conducted on entire Profimedia database using single core and 12 logical cores of one CPU respectively. Results denoted GPU1, GPU2, and GPU4 were measured on the entire database using one, two, and four GPU devices. The GPU4* results were measured on a subset of 1 million images, which were pre-cached in RAM, so the extractor was not delayed by data transactions from/to a persistent memory storage. The results are comparable since the times are normalized per signature.



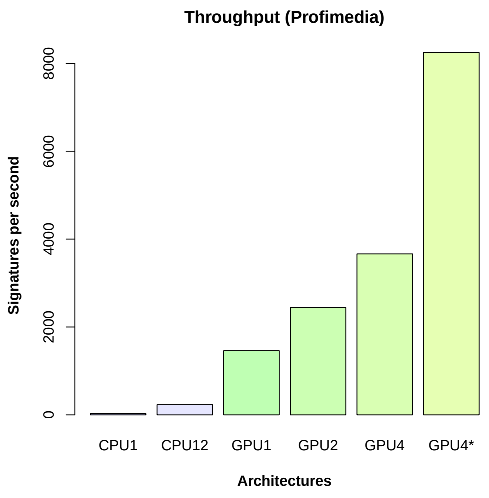


Figure 5.10: Large scale tests on Profimedia dataset

The peak performance of the extractor is 8244 signatures per second when the data are cached in RAM. Since each image has the size of approximately 33.9 KB, the system would require a persistent storage that is capable of reading data at the minimum rate of 273.3 MB/s to feed the GPUs continuously, which cannot be easily achieved by common hard disk drives. Based on the empirical data, we speculate that the feature extraction system would require at least 4 modern SSD drives connected in RAID 0 to achieve such throughput.

All the remaining experiments used two 1 TB commodity hard drives (7, 200 rpm) connected in RAID 0 to store input images and another two hard drives in RAID 0 where the extracted signatures are written. The speed of the extractor has dropped to 3661 signatures per second, when all four GPUs were utilized. Relatively speaking, the hard drives caused 2.25× drop in performance, which is quite significant. On the other hand, the CPU12 results indicate, that fully utilized Xeon achieves throughput of 303 signatures per second, thus it is hardly affected by the data loading and storing.

5.4.3 Traversing the Parameter Space

The second set of experiments was designed to search the parameter space of the extractor and find the optimal configuration for our model. The experiments were performed on the TWIC dataset, so the optimal configuration applies to this dataset alone. It is not certain yet, which parameters are data-specific and which can be used for any dataset. This problem is a topic of our future research.

Furthermore, we would like to emphasize that the amount of tests performed is far beyond the scope of this work. We have selected only a few interesting results that illustrate the search process for optimal configuration.

Evaluating Similarity Model Precision

The similarity model is evaluated against *ground truth*, an annotation created by a domain expert where images are divided into classes. Each class has one representative image, which is used as query. The other images in the same class are considered similar to the representative image. This classification is in fact a manually created clustering of the database, as images of each class are similar to each other (i.e., close from the perspective of the distance function) while images from different classes are not similar. Let us denote $GT_i \subseteq D$ the i-th class of the ground truth GT for a database D. Each image belongs to exactly one class ($GT_i \cap GT_j = \emptyset$ iff $i \neq j$) and every image is classified ($\forall i$ of $GT : \bigcup GT_i = D$). The query object of i-th class is denoted q_i^{GT} ($q_i^{GT} \in GT_i$).

For given similarity model represented by distance function d, we define an *average precision* AP_i for class GT_i as follows. Let us have a sequence of database objects $s_j^i \in D \setminus \{q_i^{GT}\}$, where $j = 1, \ldots, |D|-1$, sorted by their distance d to the query object q_i^{GT} in ascending order. In other words for $j, k \in \{1, \ldots, |D|-1\}$ and $j < k$, the $d(q_i^{GT}, s_j^i) \leq d(q_i^{GT}, s_k^i)$. The average precision is then computed as

$$AP_i = \frac{\sum_{j=1}^{|D|-1} \mathrm{hit}(s_j^i)}{|\{s_k^i : k = 1, \ldots, |D|-1 \land s_k^i \in GT_i\}|},$$

where the *hit* function is

$$\mathrm{hit}(s_j^i) = \begin{cases} \frac{|\{s_k^i : k=1,\ldots,j \land s_k^i \in GT_i\}|}{j} & \text{for } s_j^i \in GT_i, \\ 0 & \text{otherwise.} \end{cases}$$

Let us note that the average precision is equal to 1 if the sequence s_j^i contains the members of GT_i at the beginning, before any other object. The precision of the similarity model is then computed as the *mean average precision* (MAP) of the average precision of all classes. Formally, the MAP is defined as

$$MAP = \frac{1}{|GT|} \sum_{i=1}^{|GT|} AP_i$$

The mean average precision is always in the $(0, 1\rangle$ range, where values close to 0 are very poor and 1 would be achieved by an ideal similarity model. Let us emphasize that these values are also affected by the ground truth, thus the same model gets different MAP values on different datasets. Our testing dataset (TWIC) is considered rather difficult from the perspective of precision, so the MAP values above 0.3 are considered good. The most important for us is the relative change in the MAP on the same dataset for different similarity models or different model configurations.

A secondary evaluation criterium is the intrinsic dimensionality of the database (iDim). It reflects the indexability of the database by the metric access methods. The lower intrinsic dimensionality is better as the prefiltering methods can prune out more objects and save more unnecessary distance computations. If we denote X a random variable, values of which are the distances $d(o_1, o_2)$ between two randomly selected objects $o_1 \neq o_2$, the intrinsic dimensionality is computed as

$$iDim = \frac{(\mathrm{EX})^2}{2 \cdot \mathrm{Var(X)}},$$

where E stands for mean value and Var is the variance.

We take statistical approach to compute the mean and variance of X. Let us have a set $\mathcal{D} = \{d(o_1, o_2) : \forall o_1, o_2 \in D \wedge o_1 \neq o_2\}$. The mean and variance are computed as

$$\mathrm{EX} = \frac{1}{|\mathcal{D}|}\sum_{\delta \in \mathcal{D}} \delta, \quad \mathrm{Var(X)} = \frac{1}{|\mathcal{D}|}\sum_{\delta \in \mathcal{D}} (\mathrm{EX} - \delta)^2.$$

Selecting SQFD Parameters

The precision of the similarity model (i.e., the MAP and iDim results) is also affected by the selection of the α parameter and the L_p metric used in the ground distance of the SQFD. Figure 5.11 presents measured precision and intrinsic dimensionality for various L_p metrics and α values.

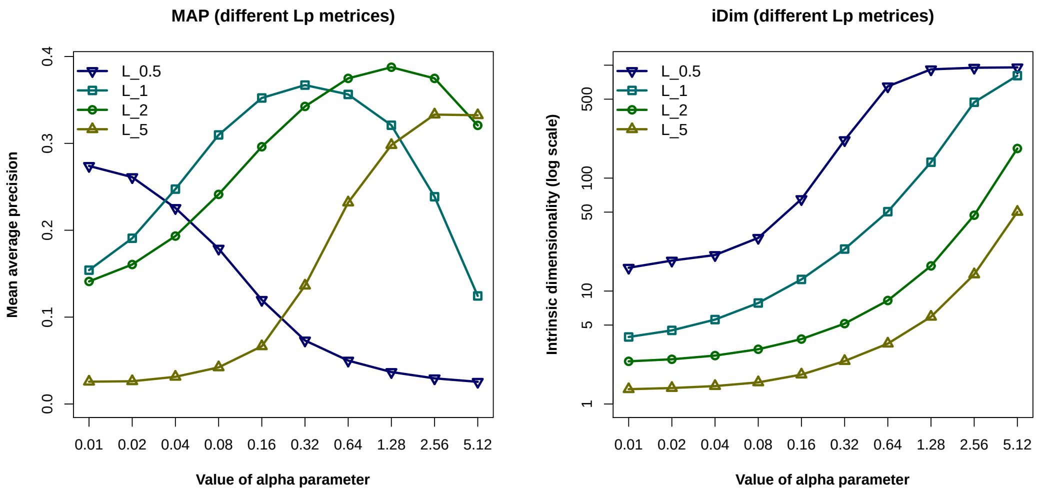


Figure 5.11: The impact of SQFD parameters on precision and iDim

As we can see, the best results are achieved for α around 1.28, with Euclidean metric L_2 in the ground distance. However, greater α also exhibit greater iDim, which is bad for indexability. For this reason, we have selected $\alpha = 0.64$ as a compromise between optimal MAP and iDim for the following experiments.

Sampling Parameters

The sampling is significantly affected by the selection of the initial set of points. Figure 5.12 presents the precision results of two sampling point sets. Both sets (*Gauss1* and *Gauss2*) are randomly generated using normal distribution with mean value of 0.5 and σ around 0.25 in both dimensions. The data were cropped to the $\langle 0, 1\rangle^2$ square by dismissing points that were out of the range. The *Gauss1* uses greater standard deviation σ, so it is closer to uniform distribution than *Gauss2*.

The results revealed two things. First of all, the sampling set that was more focused on the center of the image (*Gauss2*) has better both precision and intrinsic dimensionality. Second, at least 1,000 points need to be used to sample the image properly. A sampling set of about 2,500 points present a good compromise between precision and extraction speed as the MAP improves only slightly when more points are used.

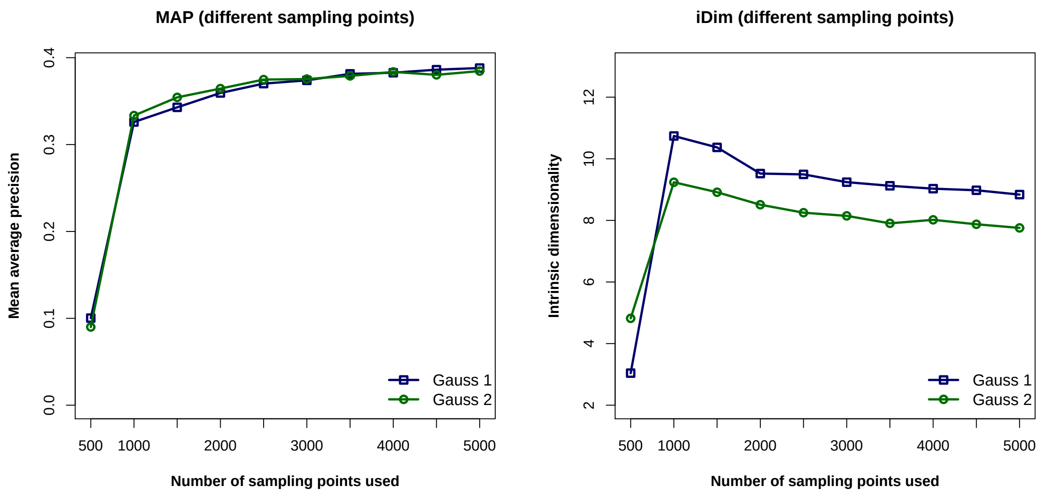


Figure 5.12: Precision of various selections of sampling point sets

Second set of tests were performed to determine the impact of greyscale size and lookup window on the quality of extracted features. Figure 5.13 shows the precision and intrinsic dimensionality of the signatures extracted with various values of G_S and r_{ce}.

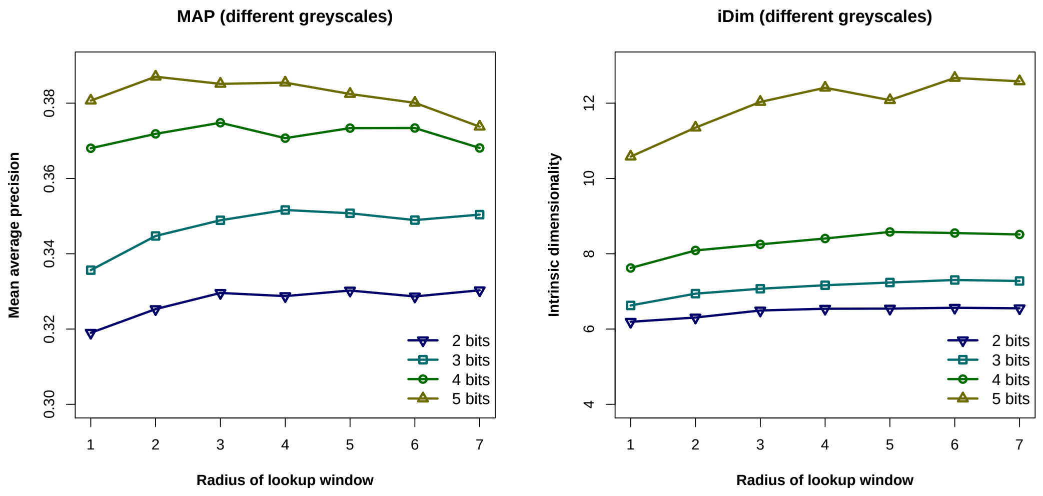


Figure 5.13: The impact of greyscale size and lookup window radius on precision

The precision rises steadily when larger greyscale is used. However, we argue that four bits ($G_S = 15$) is the best choice. The increase in MAP is rather small when more bits are used and we observe a significant rise in iDim for 5-bit greyscale. Furthermore, according to performance evaluation presented in Figure 5.8, the 4-bit configuration is still quite efficient while a significant drop in performance is observed for 5-bits and more.

The size of the lookup window have only negligible impact on precision. The best compromise in this case would be a window of 7×7 pixels ($r_{ce} = 3$) as it has the best MAP in case of 4-bit greyscale and reasonable performance according to Figure 5.8.

We have also performed tests to determine the optimal linear transformation of the feature space. These tests are to extensive to present here as they search over 7-dimensional space of multiplicative constants. The best configuration found so far was $(8, 8, 1/50, 1/50, 1/50, 2/25, 1/2)$ for the vector of (x, y, L, a, b, c, e). It achieved the mean average precision of almost 0.4. In practice, we use a combination of $(8, 8, 1/100, 1/50, 1/50, 1/25, 1/4)$, which has only slightly worse MAP, but significantly better iDim.

K-means Parameters

The most interesting results from the k-means parameters are from the tests that determine the impact of the number of iterations and the C_{min} value used for pruning small clusters. Let us revise that, clusters smaller than $C_{min} \cdot \mathcal{I}$ are dismissed in each iteration, where $\mathcal{I}$ is the number of the iteration.

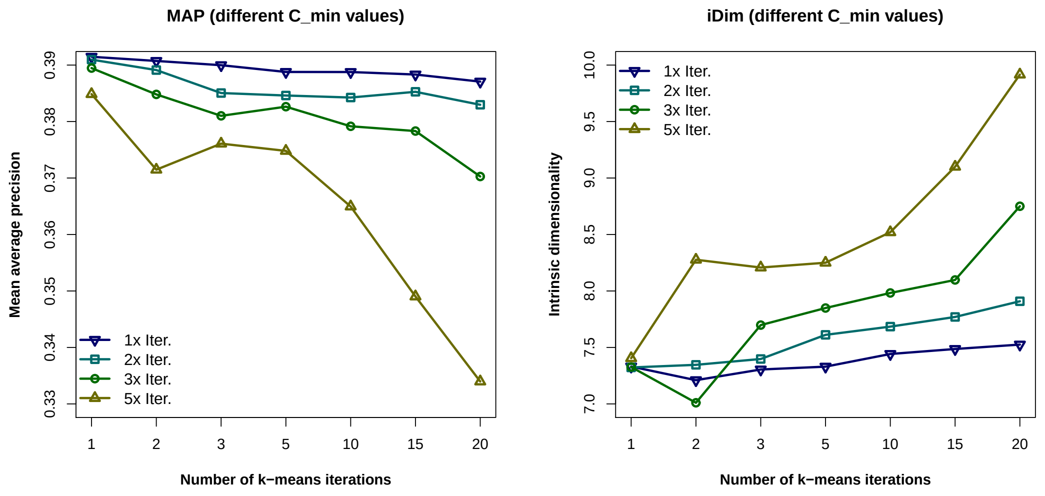


Figure 5.14: The impact of cluster pruning on precision

The results are presented in Figure 5.14. The best precision is achieved when the value of $C_{min} \cdot \mathcal{I}$ remains small in all iterations. The more iterations we perform or the larger the C_{min} parameter is, the more clusters are pruned, which does not reflect positively on the model. On the other hand, smaller signatures can be cached in larger quantities and reduce the computational time of the SQFD. As a compromise between speed and precision, the $C_{min} = 2$ and $\mathcal{I}_{max} = 10$ were selected.

The remaining parameters have only minimal impact on the precision, when they are selected reasonably. We use L_2 metric to measure distances during clustering, the joining distance $d_{min} = 0.2$, and 400 seeds as initial mean values.

Conclusion

In the conclusion, let us summarize the contributions of this thesis, put them in the context of our related research, and outline our future endeavours.

Contributions and Achieved Objectives

We have successfully met all of our outlined objectives. Our prototype implementations were proven efficient by extensive empirical tests and the GPU platform was found more than suitable for this type of problems. The proposed algorithm modifications and parallelization techniques were published in reviewed proceedings of several conferences and one impacted journal and accepted by the scientific community.

The most important contribution was made in the field of similarity search in image datasets. We have modified the SQFD distance function, which measures the dissimilarity of image feature signatures, for the GPU architecture and accelerated the similarity search process by more than two orders of magnitude. Our first results were presented at the international CIKM conference [11]. The complete work, which also included the acceleration of metric access methods, was published in the Journal of Distributed and Parallel Databases [12]. Our latest research in the field of parallel kNN queries with pivot table prefiltering is yet to be published.

The followup work on the acceleration of the extraction process, which creates feature signatures from images, achieved a similar speedup. We have successfully adopted all parts of the extraction algorithm, so it can be executed on GPUs. The parallelization of the k-means algorithm is particularly interesting, since this algorithm is used in many other data-mining problems. The proposed modifications were presented at the international Multimedia Modeling (MMM) conference [13].

In order to achieve such excellent results, we had to study the behaviour of the GPU devices. Especially, the nuances of task scheduling on CPU cores and GPU devices to ensure that the GPU does not wait for the CPU and vice versa. Our proposed solution was published at the ITAT conference [10] and implemented as an OpenCL wrapper framework, which is going to be published soon.

Related Research and Work in Progress

This thesis presents only a subset of a broad research on the topic of employing parallel architectures in large data processing, which is being conducted at the Departement of Software Engineering, Charles University in Prague. We have studied various database and big-data problems besides the similarity search.

The most challenging problems were found for the semi-structured data organized in trees (e.g., XML) and graphs (e.g., RDF). In case of XML, we have studied the possibilities of accelerating XPath queries using multi-core CPUs and NUMA servers. The work was summarized in a master thesis [142] and published at DATESO [143] and NDT [144] conferences.

As these data problems are very time consuming and share a lot of technical issues, a long-term project of developing a highly parallel framework for data processing was started at our departement. The prototype called Bobox [41, 40] is currently very competitive in the domain of data processing. It has a very sophisticated task scheduler and memory allocator [145], while it offers very simple API to the programmer. The implementation of the SPARQL query language for RDF data [65] that utilizes Bobox framework outperform other known implementations in orders of magnitude.

In the domain of relational databases, we focused on one of the most time consuming operations – the database join. We tried to accelerate the natural join operation of tables with numerical keys by the means of both multi-core CPUs and many-core GPUs. The results were presented at the DATESO conference [72]. This work was significantly extended by one of our master students and an extensive comparison of GPU hashing methods was presented in his master thesis [146].

Future Work in Similarity Search

In the future, we would like to continue the work on the similarity search and content based retrieval in image databases. The GPUs have proven more than useful for accelerating the SQFD function, so we hope they can achieve similar results for the Earth Mover's Distance function (EMD), which is even more time consuming than the SQFD. Furthermore, we would like to try to adopt other metric access methods for GPUs besides the already implemented 2-phase LAESA.

The metric indexing in a parallel environment presents far more challenges than it has been revealed. In Section 4.6, we presented the problem of kNN query with pivot table prefiltering. It has been established that any parallel solution must be suboptimal in the number of computed distances. We have proposed a novel algorithm, which achieved better results than the naïve implementation of parallel kNN, but we have tested it only on the SQFD distance function and image feature signatures. Similarity models with significantly cheaper or significantly more expensive distance functions (e.g., the structural matching of protein structures [108]) are likely to achieve better results with other approaches. Our objective is to map multiple similarity models and design a generic method of selecting the best possible parallel approach based on a cost estimation of the distance function.

Since we now posses a fast extractor and a fast similarity search engine, we are going to employ them for some real-time tasks in the computer vision. We would like to test these methods on video streams from HD camera and video streams with depth information from the Microsoft Kinect device to detect and identify predefined objects.

So far, our research on the topic of similarity search focused solely on the image data. It has been discovered that similar methods can be utilized also in the field of astrophysics. We have initiated a collaboration with a physics departement to analyze the radiation spectra of stars by the means of a parallel similarity search. Our main objective is to create a classification based solely on clustering techniques, where similar spectra are grouped to one class.

Future Work in Data Processing

We have mentioned our work in the domain of relational databases. So far, we have been able to explore the problems of hash-based joins that use numerical indices and accelerate them using a single GPU device. In the future, we would also like to research the parallelization of nested loops, hashing algorithms for string keys, and the utilization of multi-GPU configurations.

In case of semi-structured data, we have been succesful in implementing a highly optimized XPath search engine, which was capable of utilizing many CPU cores. Our approach was based on creating special linearized indices over the XML tree structures, which can be easily traversed concurrently. We hope that this approach can be adopted for the GPU implementation as well. Furthermore, we would like to test this approach on tree-like data other than XML.

The graph-like data, such as RDF or linked-data, are even more challenging. First of all, we would like to integrate the GPU devices into the Bobox framework, so we can utilize some of the existing coding, especially the SPARQL frontend. After that, we plan to design specific graph indices tailored to the GPU architecture, so we can run most of the operations on the GPU.

Most of our work was conducted on the NVIDIA GPUs with the Fermi architecture. At the end of 2012, NVIDIA released Tesla GPUs with a new architecture called Kepler. This architecture brings some significant innovations, such as the dynamic parallelism that allows the creation of nested parallel loops or a significant increase in the number of integrated cores. Furthermore, we would like to test our algorithms on the new Intel many-core architecture presented as Xeon Phi [5], which has been introduced for the general market this year.

Attached Digital Content

An optical disk with attached data is being distributed with the thesis. The disk has the following content:

- A digital representation of this thesis in PDF and PostScript (`thesis.pdf` and `thesis.ps` files in the root directory).
- LaTeX source codes for the thesis with all the figures and graphs (the `latex-src` directory). A makefile is also attached to provide a convenient way for building the thesis. The `latex-src/pic` directory contains all the embedded figures in encapsulated PostScript and SVG formats. The `latex-src/graph` directory contains all the graphs in encapsulated PostScript and their source codes for the R-Project software, which was used to create them (`http://www.r-project.org/`).
- The C++ source codes of the GPU framework (that contains the OpenCL wrapper and feeding thread pool), the prototype of the SQFD search engine and the GPU feature extractor are stored in the `src` directory. The entire code is gathered in one Visual Studio solution package, and both SQFD and feature extractor have also their own makefiles for Linux. See `src/readme.txt` for more details regarding the source codes.
- The measured results and related data are in the `data` directory. The `04-sqfd` and the `04-indexing` subdirectories contain experimental results for SQFD and indexing experiments conducted in Chapter 4. The experimental results from Chapter 5 are in `05-feature_extraction` subdirectory. Finally, the `images` directory contains the TWIC dataset and part of ALOI dataset (both images and extracted feature signatures). Each subdirectory has a `readme.txt` file, which contains more detailed information about its contents.

The digital content can also be loaded from:

`http://www.ksi.mff.cuni.cz/~krulis/thesis/`

The contents is divided into two parts, both are compressed into zip package and into tar.gz package, so they can be decompressed on Windows and on various unices conveniently.

- `thesis-dvd-content.{zip|tar.gz}` files contain everything except the images and feature signatures (i.e., the `data/images` subdirectory).
- `thesis-dvd-content-images.{zip|tar.gz}` files contain the TWIC and ALOI images and feature signatures.

Bibliography

[1] Gordon E. Moore: Moore's Law
http://en.wikipedia.org/wiki/Moore's_law

[2] Moore, G., et al.: Cramming more components onto integrated circuits. Proceedings of the IEEE **86**(1) (1998) 82–85

[3] Intel, E.: SpeedStep® Technology for the Intel® Pentium® M Processor. ftp://download.intel.com/design/network/papers/30117401.pdf (2004)

[4] Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell multiprocessor. IBM journal of Research and Development **49**(4.5) (2005) 589–604

[5] Intel: Xeon Phi Coprocessor
http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[6] JáJá, J.: An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc. (1992)

[7] Leighton, F.: Introduction to parallel algorithms and architectures. Morgan Kaufmann San Mateo, Calif. (1992)

[8] Karp, R.: A survey of parallel algorithms for shared-memory machines. (1988)

[9] Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, ACM (1967) 483–485

[10] Krulis, M., Falt, Z., Bednárek, D., Yaghob, J.: Task Scheduling in Hybrid CPU-GPU Systems. In: ITAT. (2012) 17–24

[11] Krulis, M., Lokoc, J., Beecks, C., Skopal, T., Seidl, T.: Processing the signature quadratic form distance on many-core GPU architectures. In: CIKM. (2011) 2373–2376

[12] Krulis, M., Skopal, T., Lokoc, J., Beecks, C.: Combining CPU and GPU architectures for fast similarity search. Distributed and Parallel Databases **30**(3-4) (2012) 179–207

[13] Krulis, M., Lokoc, J., Skopal, T.: Efficient Extraction of Feature Signatures Using Multi-GPU Architecture. In: Advances in Multimedia Modeling, 19th International Conference, MMM. (2013) 446–456

[14] Rojas, R.: Konrad Zuse's legacy: the architecture of the Z1 and Z3. Annals of the History of Computing, IEEE **19**(2) (1997) 5–16

[15] CRAY: The supercomputer company. http://www.cray.com

[16] Hockney, R., Jesshope, C.: Parallel Computers 2: architecture, programming and algorithms. Volume 2. Taylor & Francis (1988)

[17] Hinton, G., Sager, D., Upton, M., Boggs, D., et al.: The microarchitecture of the pentium® 4 processor. In: Intel Technology Journal, Citeseer (2001)

[18] Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technology Journal **6**(1) (2002) 4–15

[19] Koufaty, D., Marr, D.: Hyperthreading technology in the netburst microarchitecture. Micro, IEEE **23**(2) (2003) 56–65

[20] Butler, M., Barnes, L., Sarma, D., Gelinas, B.: Bulldozer: An approach to multithreaded compute performance. Micro, IEEE **31**(2) (2011) 6–15

[21] Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2005)

[22] Tanenbaum, A.: Modern operating systems. Volume 2. prentice Hall New Jersey (1992)

[23] Silberschatz, A., Galvin, P., Gagne, G.: Applied operating system concepts. John Wiley & Sons, Inc. (2001)

[24] Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) **9**(1) (1991) 21–65

[25] Intel: IA-32 Intel Architecture Software Developers Manual
`http://download.intel.com/products/processor/manual/325462.pdf`

[26] NUMA: Non-Uniform Memory Architecture
`http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access`

[27] Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Incorporated (2007)

[28] Intel: Threading Building Blocks Documentation
`http://threadingbuildingblocks.org/documentation.php`

[29] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel programming in OpenMP. Morgan Kaufmann (2000)

[30] OpenMP: The API Specification for Parallel Programming. `www.openmp.org`

[31] NVIDIA: Fermi GPU Architecture
`http://www.nvidia.com/object/fermi-architecture.html`

[32] NVIDIA: Kepler GPU Architecture
`http://www.nvidia.com/object/nvidia-kepler.html`

[33] NVIDIA: CUDA C Programming Guide
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[34] NVIDIA: CUDA C Best Practices Guide
`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`

[35] Kirk, D., Wen-mei, W.: Programming massively parallel processors: a hands-on approach. Morgan Kaufmann (2010)

[36] NVIDIA: CUDA. `http://www.nvidia.com/object/cuda_home_new.html`

[37] AMD: Accelerated Parallel Processing SDK
`http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`

[38] Khronos: OpenCL – The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`

[39] ISO: IEC 9899: 1999, Programming Languages–C. International Organization for Standardization (1999)

[40] Bobox: A highly parallel framework for data processing
`http://www.ksi.mff.cuni.cz/en/sw/bobox.html`

[41] Bednárek, D., Dokulil, J., Yaghob, J., Zavoral, F.: Bobox: Parallelization Framework for Data Processing. Advances in Information Technology and Applied Computing (2012) in press

[42] Sinnen, O.: Task scheduling for parallel systems. Volume 60. Wiley-Interscience (2007)

[43] El-Rewini, H., Lewis, T., Ali, H.: Task scheduling in parallel and distributed systems. Prentice-Hall, Inc. (1994)

[44] Shirazi, B., Kavi, K., Hurson, A.: Scheduling and load balancing in parallel and distributed systems. IEEE Computer Society Press (1995)

[45] Bruno, J., Coffman Jr, E., Sethi, R.: Scheduling independent tasks to reduce mean finishing time. Communications of the ACM **17**(7) (1974) 382–387

[46] Lenstra, J., Kan, A., Brucker, P.: Complexity of machine scheduling problems. (1977)

[47] Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc. (1990)

[48] Chekuri, C., Khanna, S.: A polynomial time approximation scheme for the multiple knapsack problem. SIAM Journal on Computing **35**(3) (2005) 713–728

[49] Mounie, G., Rapine, C., Trystram, D.: Efficient approximation algorithms for scheduling malleable tasks. In: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, ACM (1999) 23–32

[50] Kelley, J.: The critical-path method: Resources planning and scheduling. Industrial scheduling (1963) 347–365

[51] Stankovic, J.: Deadline scheduling for real-time systems: EDF and related algorithms. Springer (1998)

[52] Krishna, C.: Real-Time Systems. Wiley Online Library (1999)

[53] Kuo, S., Lee, B., Tian, W.: Real-time digital signal processing: implementations and applications. Wiley (2006)

[54] Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM) **20**(1) (1973) 46–61

[55] Shirazi, B., Wang, M., Pathak, G.: Analysis and evaluation of heuristic methods for static task scheduling. Journal of Parallel and Distributed Computing **10**(3) (1990) 222–232

[56] Iverson, M., Ozguner, F.: Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In: Heterogeneous Computing Workshop, 1998.(HCW 98) Proceedings. 1998 Seventh, IEEE (1998) 70–78

[57] Rotithor, H.: Taxonomy of dynamic task scheduling schemes in distributed computing systems. In: Computers and Digital Techniques, IEE Proceedings-. Volume 141., IET (1994) 1–10

[58] Lucco, S.: A dynamic scheduling method for irregular parallel programs. In: ACM SIGPLAN Notices. Volume 27., ACM (1992) 200–211

[59] Wang, L., Huang, Y., Chen, X., Zhang, C.: Task scheduling of parallel processing in CPU-GPU collaborative environment. In: Computer Science and Information Technology, 2008. ICCSIT'08. International Conference on, IEEE (2008) 228–232

[60] Jablin, T., Prabhu, P., Jablin, J., Johnson, N., Beard, S., August, D.: Automatic CPU-GPU communication management and optimization. In: ACM SIGPLAN Notices. Volume 46., ACM (2011) 142–151

[61] Jimenez, V., Vilanova, L., Gelado, I., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. High Performance Embedded Architectures and Compilers (2009) 19–33

[62] Shirahata, K., Sato, H., Matsuoka, S.: Hybrid map task scheduling for gpu-based heterogeneous clusters. In: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE (2010) 733–740

[63] Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113

[64] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198

[65] Falt, Z., Bednarek, D., Cermak, M., Zavoral, F.: On Parallel Evaluation of SPARQL Queries. In: DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications. (2012) 97–102

[66] Cermak, M., Falt, Z., Dokulil, J., Zavoral, F.: SPARQL Query Processing Using Bobox Framework. In: SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing. (2011) 104–109

[67] Lassila, O., Swick, R., et al.: Resource description framework (RDF) model and syntax specification. (1998)

[68] Bednárek, D.: Output-driven XQuery evaluation. Intelligent Distributed Computing, Systems and Applications (2008) 55–64

[69] Bednarek, D., Dokulil, J.: Tri Query: Modifying XQuery for RDF and Relational Data. In: Database and Expert Systems Applications (DEXA), 2010 Workshop on, IEEE (2010) 342–346

[70] Falt, Z., Yaghob, J.: Task Scheduling in Data Stream Processing. In: Proceedings of the Dateso 2011 Workshop. (2011) 85–96

[71] Fibers: Lighweight threads of execution
`http://en.wikipedia.org/wiki/Fiber_(computer_science)`

[72] Krulis, M., Yaghob, J.: Revision of Relational Joins for Multi-Core and Many-Core Architectures. In: DATESO. (2011) 229–240

[73] LCG: Linear congruential generator
`http://en.wikipedia.org/wiki/Linear_congruential_generator`

[74] Datta, R., Joshi, D., Li, J., Wang, J.Z.: Image retrieval: Ideas, influences, and trends of the new age. ACM Computing Surveys (CSUR) **40**(2) (2008) 5

[75] Lokoč, J., Grošup, T., Skopal, T.: Image exploration using online feature extraction and reranking. In: Proceedings of the 2nd ACM International Conference on Multimedia Retrieval, ACM (2012) 66

[76] Lokoč, J., Grošup, T., Skopal, T.: SIR: the smart image retrieval engine. Similarity Search and Applications (2012) 240–241

[77] Beecks, C., Uysal, M., Seidl, T.: A comparative study of similarity measures for content-based multimedia retrieval. In: Multimedia and Expo (ICME), 2010 IEEE International Conference on, IEEE (2010) 1552–1557

[78] Csurka, G., Dance, C., Fan, L., Willamowski, J., Bray, C.: Visual categorization with bags of keypoints. In: Workshop on statistical learning in computer vision, ECCV. Volume 1. (2004) 22

[79] Lowe, D.G.: Object recognition from local scale-invariant features. In: Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on. Volume 2., Ieee (1999) 1150–1157

[80] Salembier, P., Sikora, T., Manjunath, B.: Introduction to MPEG-7: multimedia content description interface. John Wiley & Sons, Inc. (2002)

[81] McLaren, K.: XIII—The Development of the CIE 1976 (L* a* b*) Uniform Colour Space and Colour-difference Formula. Journal of the Society of Dyers and Colourists **92**(9) (1976) 338–341

[82] Huttenlocher, D., Klanderman, G., Rucklidge, W.: Comparing images using the Hausdorff distance. Pattern Analysis and Machine Intelligence, IEEE Transactions on **15**(9) (1993) 850–863

[83] Beecks, C., Uysal, M., Seidl, T.: Signature quadratic form distance. In: Proceedings of the ACM International Conference on Image and Video Retrieval, ACM (2010) 438–445

[84] Beecks, C., Ivanescu, A., Kirchhoff, S., Seidl, T.: Modeling image similarity by gaussian mixture models and the signature quadratic form distance. In: Computer Vision (ICCV), 2011 IEEE International Conference on, IEEE (2011) 1754–1761

[85] Rubner, Y., Tomasi, C., Guibas, L.: The earth mover's distance as a metric for image retrieval. International Journal of Computer Vision **40**(2) (2000) 99–121

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

[87] Beecks, C., Uysal, M., Seidl, T.: Efficient k-nearest neighbor queries with the signature quadratic form distance. In: Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on, IEEE (2010) 10–15

[88] Hafner, J., Sawhney, H., Equitz, W., Flickner, M., Niblack, W.: Efficient color histogram indexing for quadratic form distance functions. Pattern Analysis and Machine Intelligence, IEEE Transactions on **17**(7) (1995) 729–736

[89] Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys (CSUR) **33**(3) (2001) 273–321

[90] Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Volume 32 of Advances in Database Systems. Springer (2006)

[91] Skopal, T., Bustos, B.: On nonmetric similarity search problems in complex domains. ACM Computing Surveys (CSUR) **43**(4) (2011) 34

[92] Skopal, T., Lokoč, J.: NM-tree: Flexible approximate similarity search in metric and non-metric spaces. In: Database and Expert Systems Applications, Springer (2008) 312–325

[93] Vidal Ruiz, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters **4**(3) (1986) 145–157

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

[95] Chávez, E., Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on, IEEE (2000) 75–86

[96] Knuth, D.E.: The art of computer programming, Vol. 3. Volume 109. Addison-Wesley, Reading, MA (1973)

[97] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. Volume 19. ACM (1990)

[98] Ciaccia, P., Patella, M., Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proceedings of the Twenty-third International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann (1997) 426–435

[99] Skopal, T., Pokorný, J., Snasel, V.: PM-tree: Pivoting metric tree for similarity search in multimedia databases. In: ADBIS (Local Proceedings). (2004)

[100] Novak, D., Batko, M.: Metric index: An efficient and scalable solution for similarity search. In: Similarity Search and Applications, 2009. SISAP'09. Second International Workshop on, IEEE (2009) 65–73

[101] Jagadish, H., Ooi, B., Tan, K., Yu, C., Zhang, R.: iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS) **30**(2) (2005) 364–397

[102] Hetland, M.: Ptolemaic indexing. arXiv preprint arXiv:0911.4384 (2009)

[103] Lokoč, J., Hetland, M., Skopal, T., Beecks, C.: Ptolemaic indexing of the signature quadratic form distance. In: Proceedings of the Fourth International Conference on SImilarity Search and APplications, ACM (2011) 9–16

[104] Mathies, T.R.: A fast parallel algorithm to determine edit distance. (1988)

[105] Galper, A.R., Brutlag, D.L.: Parallel similarity search and alignment with the dynamic programming method. Knowledge Systems Laboratory, Medical Computer Science, Stanford University (1990)

[106] Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of the tenth annual ACM symposium on Theory of computing, ACM (1978) 114–118

[107] Gish, W., States, D.J.: Identification of protein coding regions by database similarity search. Nature genetics **3**(3) (1993) 266–272

[108] Galgonek, J., Kruliš, M., Hoksza, D.: On the Parallelization of the SProt Measure and the TM-score Algorithm. In: 3rd International Workshop on High Performance Bioinformatics and Biomedicine (HiBB). (2012)

[109] Alpkocak, A., Danisman, T., Tuba, U.: A parallel similarity search in high dimensional metric space using m-tree. Advanced Environments, Tools, and Applications for Cluster Computing (2002) 247–252

[110] Batko, M., Gennaro, C., Savino, P., Zezula, P.: Scalable similarity search in metric spaces. In: Proceedings of the DELOS Workshop on Digital Library Architectures: Peer-to-Peer, Grid, and Service-Orientation, Edizioni Libreria Progetto, Padova, Italy. (2004) 213–224

[111] Novak, D., Zezula, P.: M-chord: a scalable distributed similarity search structure. In: Proceedings of the 1st international conference on Scalable information systems, ACM (2006) 19

[112] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review **31**(4) (2001) 149–160

[113] Berchtold, S., Böhm, C., Braunmüller, B., Keim, D.A., Kriegel, H.P.: Fast parallel similarity search in multimedia databases. Volume 26. ACM (1997)

[114] Bustos, B., Deussen, O., Hiller, S., Keim, D.: A graphics hardware accelerated algorithm for nearest neighbor search. Computational Science–ICCS 2006 (2006) 196–199

[115] Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE (2008) 1–6

[116] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM (JACM) **45**(6) (1998) 891–923

[117] Barrientos, R., Gómez, J., Tenllado, C., Prieto, M.: Heap based k-nearest neighbor search on gpus. In: Congreso Espanol de Informática (CEDI). (2010) 559–566

[118] Geusebroek, J.M., Burghouts, G.J., Smeulders, A.W.: The amsterdam library of object images. International Journal of Computer Vision **61**(1) (2005) 103–112

[119] Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: CoPhIR: a test collection for content-based image retrieval. arXiv preprint arXiv:0905.4627 (2009)

[120] Profimedia: Image Database. `http://www.profimedia.cz/`

[121] Beecks, C., Lokoč, J., Seidl, T., Skopal, T.: Indexing the signature quadratic form distance for efficient content-based multimedia retrieval. In: Proceedings of the 1st ACM International Conference on Multimedia Retrieval, ACM (2011) 24

[122] Colantoni, P., Boukala, N., Da Rugna, J.: Fast and accurate color image processing using 3d graphics cards. In: 8th International Fall Workshop-Vision Modeling and Visualization 2003, Proceedings November 19-21, 2003, München, Germany. (2003) 383–390

[123] Fung, J.: Computer Vision on the GPU. GPU Gems **2** (2005) 649–666

[124] Canny, J.: A computational approach to edge detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on (6) (1986) 679–698

[125] Roodt, Y., Visser, W., Clarke, W.: Image processing on the GPU: Implementing the Canny edge detection algorithm. In: Symp. Pattern Recognition Association of South Africa. (2007)

[126] Luo, Y., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE (2008) 1–8

[127] Ogawa, K., Ito, Y., Nakano, K.: Efficient canny edge detection using a gpu. In: Networking and Computing (ICNC), 2010 First International Conference on, IEEE (2010) 279–280

[128] Heymann, S., Muller, K., Smolic, A., Frohlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision. (2007) 144

[129] Li, X., Fang, Z.: Parallel clustering algorithms. Parallel Computing **11**(3) (1989) 275–290

[130] Shalom, S., Dash, M., Tue, M.: Efficient k-means clustering using accelerated graphics processors. Data Warehousing and Knowledge Discovery (2008) 166–175

[131] Farivar, R., Rebolledo, D., Chan, E., Campbell, R.: A parallel implementation of k-means clustering on GPUs. In: Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). (2008) 340–345

[132] Hong-Tao, B., Li-li, H., Dan-tong, O., Zhan-shan, L., He, L.: K-means on commodity GPUs with CUDA. In: Computer Science and Information Engineering, 2009 WRI World Congress on. Volume 3., IEEE (2009) 651–655

[133] Zechner, M., Granitzer, M.: Accelerating k-means on the graphics processor via CUDA. In: Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on, IEEE (2009) 7–15

[134] Parker, J.R.: Algorithms for image processing and computer vision. Wiley Publishing (2010)

[135] Tkalcic, M., Tasic, J.F.: Colour spaces: perceptual, historical and applicational background. Volume 1. IEEE (2003)

[136] Gotlieb, C.C., Kreyszig, H.E.: Texture descriptors based on co-occurrence matrices. Computer Vision, Graphics, and Image Processing **51**(1) (1990) 70–86

[137] Hartigan, J.A., Wong, M.A.: Algorithm AS 136: A k-means clustering algorithm. Applied statistics (1979) 100–108

[138] Dehne, F., Noltemeier, H.: Voronoi trees and clustering problems. Information Systems **12**(2) (1987) 171–175

[139] SIRET: SImilarity RETrieval Research Group, Charles University in Prague `http://www.siret.cz/`

[140] Pelleg, D., Moore, A.: Accelerating exact k-means algorithms with geometric reasoning. In: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (1999) 277–281

[141] Lokoč, J., Novák, D., Skopal, T., Sibirkina, N.: Thematic Web Images Collection, SIRET Research Group, `http://siret.ms.mff.cuni.cz/twic` (2012)

[142] Kruliš, M.: Algorithms for Parallel Searching in XML Datasets. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague (2009)

[143] Kruliš, M., Yaghob, J.: Efficient Implementation of XPath Processor on Multi-Core CPUs. In: DATESO. (2010) 60–71

[144] Kruliš, M., Yaghob, J.: The Location Path to Hell Is Paved With Unoptimized Axes: XPath Implementation Tips. In: Networked Digital Technologies. Springer (2010) 474–487

[145] Falt, Z.: Scheduler and memory allocator for the Bobox system. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague (2010)

[146] Vansa, R.: Parallel Data-processing on GPGPU. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague (2012)