A Study on GPU-Accelerated Data Analysis Techniques

September 2016

Yusuke Kozawa

A Study on GPU-Accelerated Data Analysis Techniques

Graduate School of Systems and Information Engineering
University of Tsukuba

September 2016

Yusuke Kozawa

Abstract

Rapid growth of data volumes necessitates efficient analysis techniques for large-scale data. To this end, GPUs (graphics processing units) can be considered a cost-effective means to accelerate data analysis techniques. GPUs have recently evolved as many-core accelerators and shown superior performance over ordinary CPUs. However, in order to harness the power of GPUs, we need to take into account several characteristics of GPUs.

The main challenges of using GPUs are three-fold: massive and hierarchical parallelism, memory hierarchy, and load balancing. The first challenge means that GPUs accommodate thousands of threads and the threads are hierarchically organized. In this situation, conventional multi-threaded algorithms for CPUs cannot be directly applied to GPUs. The second challenge is to utilize memory hierarchy available on GPUs. Various kinds of memory have different properties (e.g., bandwidth and size), and exploiting them effectively is important for achieving high performance. The third challenge is to balance workloads processed by massive and hierarchical threads. This is especially important when skewed data, which is common in the real world, is handled.

This dissertation explores efficient and effective ways to exploit GPUs by inspecting four problems: frequent itemset mining from uncertain data, comparison sorting, canopy clustering, and graph clustering. To make the best use of GPUs, the following three techniques are exploited: (1) effective data structures, (2) data-parallel primitives, and (3) capturing key-component on specific algorithms. Through experimental evaluation, this dissertation reveals that proposed methods substantially outperform existing GPU-based methods and CPU implementations. Concretely, we obtained the following results: GPU-accelerated frequent itemset mining from uncertain data outperformed a CPU parallel implementation by a factor of up to 5.5; our comparison sorting method improved throughput by 32% over an existing GPU-based method; canopy clustering on a GPU was at most 2.5 times faster than CPU parallel implementation, even if the CPU counterparts used two octa-core processors; and GPU-accelerated label propagation achieved 30 times higher performance on average than a CPU implementation of label propagation.

Acknowledgements

In the course of working on this dissertation, I have received considerable support from many people. First of all, I would like to express gratitude to my supervisor, Professor Hiroyuki Kitagawa, for his helpful, insightful, and invaluable comments on this work. Without his advice, this work cannot be accomplished. I am also sincerely grateful to Associate Processor Toshiyuki Amagasa for supportive and constructive comments and suggestions. I would also like to thank Assistant Professor Yasuhiro Hayase, Assistant Professor Chiemi Watanabe, and Assistant Professor Hiroaki Shiokawa. They considerably helped me from various aspects.

The following committee members of this doctoral dissertation generously took the time to examine this work and make suggestive comments: Professor Hiroyuki Kitagawa, Professor Jun Miyazaki, Professor Kouichi Wada, Professor Daisuke Takahashi, and Associate Professor Toshiyuki Amagasa. Their comments greatly helped me to improve the quality of this dissertation.

The members of Kitagawa Data Engineering Laboratory were also very helpful throughout working on this dissertation. The discussion with them were highly instructive and chitchat also encouraged me to continue to undertake this research. I am also indebted to Ms. Yumiko Hisamatsu and Ms. Tetsuko Sato, for their assistance of clerical work.

Last but not least, my parents, Hisashi and Mitsue, have been patient and supportive throughout the doctoral program. It would have been impossible to complete this dissertation without their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances of technology lead to the growing amount of available data in the real world, and analyzing such large-scale data in realistic time requires efficient analysis techniques. Parallel computing is a common means to accelerate analyses by performing computations simultaneously on one or more computers. Recently, in the fields of parallel computing and high performance computing, *heterogeneous computing* has attracted significant attention because it can achieve both superior performance and lower power consumption than common “homogeneous” systems [95]. Heterogeneous computing refers to the system that not only uses CPUs but also is equipped with other kinds of co-processors such as GPUs (graphics processing units). Since CPUs and co-processors have different strengths and weaknesses, heterogeneous systems are capable of achieving computational gains by assigning workload to an appropriate processor. Heterogeneous computing has been increasingly important, as can be seen from the fact that many supercomputers of TOP500[1] and Green500[2] utilize CPUs and co-processors.

Among such co-processors, GPUs and Intel® Xeon Phi™ are more commonly employed. These co-processors fall under the category of *many-core* processors, which contain tens to thousands of processing units. For instance, NVIDIA Tesla K40 GPU[3] consists of 2,880 simple processing units, and Intel® Xeon Phi™ Processor 7120p[4] includes 61 relatively complex cores. Compared to Intel® Xeon Phi™, GPUs excel in computing massive amounts of simple and independent computations. This means that GPUs are suitable for highly data-parallel processing. Many

[1]`http://www.top500.org/`

[2]`http://www.green500.org/`

[3]`http://www.nvidia.com/object/tesla-workstations.html`

[4]`http://ark.intel.com/products/75799/`

data analysis techniques tend to proceed in a data-parallel manner, and the available parallelism grows as the data size increases. Therefore, GPUs are considered as a more promising solution than Intel® Xeon Phi™ for accelerating large-scale data analysis techniques. Thus this work focuses on GPU-accelerated data analysis techniques.

As mentioned above, GPUs contain a large number of simple processing units. By exploiting this massive parallelism, GPUs have recently been extensively utilized for accelerating a wide range of applications [104]. For instance, GPUs have accelerated database processing such as sorting [8, 73] and data mining techniques such as clustering [69, 80, 133]. However, in order to harness the power of GPUs, we need to take into account several characteristics of GPUs. The main challenges of using GPUs are three-fold: (1) massive and hierarchical parallelism, (2) memory hierarchy, and (3) load balancing. The first challenge means that GPUs accommodate thousands of threads and the threads are hierarchically organized. In this situation, conventional multi-threaded algorithms for CPUs cannot be directly applied to GPUs, because they are commonly run by up to tens of non-hierarchical threads. The second challenge is to utilize memory hierarchy available on GPUs. Various kinds of memory have different properties (e.g., bandwidth and size), and exploiting them effectively is important for achieving high performance. The third challenge is to balance workloads processed by massive and hierarchical threads. This is especially important when skewed data, which is common in the real world, is handled.

This dissertation presents efficient and effective ways to utilize GPUs by inspecting a broad spectrum of problems and algorithms. Specifically, the following three techniques are exploited for addressing the above challenges:

- ***Effective data structures***: Available parallelism and usage of memory hierarchy heavily depends on not only algorithm but also underlying data structures. Thus, we carefully design the data structures used in specific algorithms so that efficient parallel processing and memory accesses are allowed. Consequently, we can achieve high effective memory bandwidth of GPUs.

- ***Data parallel primitives***: Primitives are the operations that commonly appears in data-parallel computing, and they can be efficiently implemented on GPUs. By transforming algorithms into a combination of such primitives, we can harness the maximum power of GPUs.

- ***Capturing key components***: Algorithms usually have a key component, the dominant part of overall computational cost. By carefully capturing and parallelizing it for GPUs, we can achieve order-of-magnitude speedups.

On the basis of the techniques, this dissertation accelerates processing of four problems: (1) frequent itemset mining from uncertain data, (2) comparison sorting, (3) canopy clustering, and (4) graph clustering.

Frequent itemset mining from uncertain data. Frequent itemset mining [3] from uncertain data is one of the major research issues in the area of uncertain data management, and there have been a number of algorithms [17, 76, 124]. However, existing algorithms suffer from high computational cost for dealing with uncertainty, such as probability computation. To mitigate this problem, we accelerate the computation by using GPUs, with effective data structures and data parallel primitives.

Comparison sorting. Sorting is a fundamental operation in computer science, especially database systems, and its acceleration has significant importance [68]. In particular, comparison sort is a sorting algorithm that determines the order of elements on the basis of comparison operations, and it has the advantage that it can be applied to any kind of data in principle if comparison is possible. In this dissertation, we develop a method based on two existing algorithms, samplesort [73] and merge sort [8]. This method takes into account efficient access patterns to GPU memory and also utilizes data parallel primitives, thereby achieving high throughput.

Canopy clustering. Canopy clustering is a preprocessing method for standard clustering algorithms such as *k*-means and hierarchical agglomerative clustering [90]. Canopy clustering can greatly reduce the computational cost of clustering algorithms. However, canopy clustering itself may also take a vast amount of time for handling massive data. To address this problem, this dissertation presents efficient algorithms and implementations of canopy clustering on GPUs. We not only accelerate the computation of original canopy clustering, but also propose an algorithm using grid index, by exploiting effective data structures and data parallel primitives.

Graph clustering. Graph clustering, also known as community detection, is one of the prominent techniques for graph data analytics, gaining much attention from many researchers and developers [35]. However, there is a problem that the size of available graph data becomes increasingly gigantic. To tackle this problem, this dissertation proposes an algorithm based on label propagation [112], which is known as one of the fastest graph clustering algorithms. Our algorithm consists of multiple data parallel primitives and also takes into account load balancing by using the primitives whose performance is not largely affected by the existence of skewness, thereby enabling efficient processing of skewed graph data, which is common in the real world.

Experiments confirmed that our proposals outperform existing solutions. Concretely, we obtained the following results:

- GPU-accelerated frequent itemset mining from uncertain data outperformed a CPU parallel implementation by a factor of up to 5.5.
- Our comparison sorting method improved throughput by 32% over an existing GPU-based method.
- Canopy clustering on a GPU was at most 2.5 times faster than CPU parallel implementation, even if the CPU counterparts used two octa-core processors.
- GPU-accelerated label propagation achieved 30 times higher performance on average than a CPU implementation of label propagation.

From these results, we believe that our GPU-acceleration techniques help to improve the performance of not only the specific four problems but also a wider range of applications.

Organization. The rest of this dissertation is organized as follows. Chapter 2 discusses related work, including GPU computing, frequent itemset mining, sorting, clustering, and graph clustering. Chapters 3–6 describe GPU-accelerated analysis techniques on the four problems, respectively. Specifically, Chapter 3 presents a GPU-based method of frequent itemset mining from uncertain data. Chapter 4 explains a novel parallel comparison-sorting method for GPUs. Chapter 5 provides GPU-accelerated canopy clustering, including GPU parallelization of original canopy clustering and the use of grid index. Chapter 6 details GPU-accelerated label propagation for graph clustering. Finally, Chapter 7 concludes this dissertation with possible future directions.

Chapter 2

Related Work

This chapter discusses related work of this study. Section 2.1 overviews preliminaries and related work on GPU computing, including the CUDA framework and data-parallel primitives. Section 2.2 gives an overview of research work using GPUs to accelerate various kinds of data analysis techniques in the fields of databases and data mining. Sections 2.3–2.6 reviews related work on the specific four problems, namely frequent itemset mining, sorting, clustering, and graph clustering. Section 2.7 discusses the positions of proposed methods and also mentions Intel® Xeon Phi™.

2.1 GPU computing

GPU computing means the use of GPUs for accelerating general-purpose computations rather than graphics tasks, which are the original targets of GPUs [104]. GPUs have recently evolved as many-core processors and have been utilized to accelerate a wide range of applications such as data analytics and scientific computations. The advantages of GPUs over traditional CPUs are their high computational performance with a large number of simple processing units, relatively low price, and low power consumption. In order to develop programs for GPUs, the de-facto standard framework is *CUDA (compute unified device architecture)* provided by NVIDIA [98]. The rest of this section firstly describes the architectural features of GPUs and programming model of CUDA in Section 2.1.1 *Data-parallel primitives* implemented on CUDA, which are extensively utilized in this study, are summarized in Section 2.1.2.

2.1.1 CUDA

The CUDA GPU architecture is made up of multiple *streaming multiprocessors (SMs)*, which in turn consist of many simple processing units called *scalar processors (SPs)*. CUDA provides fine-grained parallelism by launching a massive number of lightweight threads. A large number of threads are grouped into a *thread block* (or a *block* for short). Typical numbers of threads per block are 128 and 256. Threads within a block run concurrently on an SM, sharing the resources of the SM. On the other hand, an SM can accommodate multiple blocks simultaneously, maintaining its resources among blocks and scheduling the threads of blocks. Blocks comprise a *grid*, which is generated each time when functions to be executed on GPUs are called; such functions are referred to as *kernels*.

SMs employ an architecture called *SIMT (single-instruction, multiple-thread)* to efficiently manage a large number of threads. Threads on SMs are managed and scheduled in groups of 32 parallel threads called *warps*, and multiple warps form a block. A warp carries out a common instruction at a time, and thus the most efficient case is achieved if all threads of a warp follow the same execution path. If threads of a warp follow different paths, then the warp serially executes each path in turn, performing redundant operations consequently.

Meanwhile, GPUs have several kinds of memory. The largest memory on GPUs is *global memory*. For instance, NVIDIA Tesla K40 has the global memory of 12 GB. It can be accessible from all threads of a grid and has a high bandwidth although the access latency is high. SMs also contain local memory, which can be accessed much faster than global memory although the size is small (up to 96 KB, depending on the microarchitecture of GPUs). This memory can be used as *shared memory*, which is shared by threads within a block and can be used to exchange data among the threads.

There are important accessing patterns to global memory, called *coalesced accesses* [98]. In general, global memory accesses are serviced with one or more memory transactions, depending on the access pattern. If all threads of a warp access an aligned and contiguous region of 128 bytes, then these accesses are coalesced into one transaction. Otherwise, the accesses are partly coalesced into not one but multiple transactions and the performance deteriorates compared to the single-transaction case.

2.1.2 Data-parallel primitives

Data-parallel primitives are basic operations in data-parallel computing [56]. They can be used as building blocks to efficiently parallelize more complex algorithms, because implementations of primitives can be highly tuned and deliver high performance. GPU implementations of primitives have been explored since the introduction of CUDA [46, 50, 51, 53, 54, 56, 139]. This subsection summarizes primitives commonly used in this dissertation and related work on the primitives if any.

Map

A *map* operation takes an array $[a_0, a_1, \ldots, a_{n-1}]$ of n elements and a unary function f, and returns the array $[f(a_0), f(a_1), \ldots, f(a_{n-1})]$. This operation can be easily parallelized by assigning each thread to each element.

Reduce

A *reduce* operation takes an array $[a_0, a_1, \ldots, a_{n-1}]$ of n elements and a binary associative and commutative operator $\oplus$, and returns the value $a_0 \oplus \cdots \oplus a_{n-1}$. For example, if the binary operator is addition, the reduce operation computes the sum over the array. Several implementations of reduce can be possible, and, among them, a tree-based approach presented by NVDIIA [50] is commonly used.

Scan

A *scan*, also known as *prefix sum*, operation takes an array $[a_0, a_1, \ldots, a_{n-1}]$ of n elements and a binary associative and commutative operator $\oplus$ with identity I, and returns the array $[I, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}]$. This operation is a useful building block for other primitives and many algorithms such as sort and breadth first search [139].

While the serial algorithm is simple, an efficient parallel implementation is surprisingly difficult. Thus much effort has been made to develop efficient scan methods for GPUs [46,51,139]. The pioneering work on CUDA was done by Harris [51]. He develop a work-efficient method based on the concept of a balanced binary tree. More recently, Ha and Han [46] presented a novel scalable parallel scan method based on two techniques: (1) work-efficient and depth-optimal intra-block scan and (2) memory bandwidth efficient global decomposition. The current state of the art is considered to be the method, called *StreamScan*, proposed by Yan et al. [139]. While conventional scan implementations require three kernels and at least $3n$ (n

is the problem size) global memory accesses, their method realizes the scan operation with only one kernel and $2n$ global memory accesses, thereby achieving large performance improvements.

Filter

A *filter* operation selects a subset of elements from an array according to a unary predicate P (i.e., a function that takes one argument and returns zero or one) [56]. This operation can be implemented with the map and scan primitives in three steps. First, the map primitive with predicate P is applied to the input array and produces a corresponding array `flag` of boolean values. Second, we perform scan with addition on the array `flag` and store the result into another array `index`. Finally, the elements that satisfy the predicate P are outputted based on the two arrays. If `flag`$[i]$ is one, the ith element of the input array is moved to the output array at the index `index`$[i]$. Otherwise, the element is ignored and filtered out.

Gather and scatter

A *gather* operation carries out indexed reads from an array [53, 56]. It takes two arrays A of n elements and B of m elements and returns the array C of m elements that are computed as $C[i] \leftarrow A[B[i]]$. A *scatter* operation is dual to gather, performing indexed writes to an array [53, 56]. It takes two arrays A of n elements and B of m elements and returns the array C of m elements that are computed as $C[B[i]] \leftarrow A[i]$.

Segmented sort and segmented reduce

Segmented sort and *segmented reduce* are parts of the library *Modern GPU* [8], efficiently working on multiple irregular-length segments within one array in parallel. A segmented sort operation takes two arrays $A = [a_0, a_1, \ldots, a_{n-1}]$ and $S = [s_0, s_1, \ldots, s_{m-1}, s_m = n]$, and sorts each subarray $[a_{s_i}, \ldots, a_{s_{i+1}-1}]$ of A. A segmented reduce operation takes two arrays $A = [a_0, a_1, \ldots, a_{n-1}]$ and $S = [s_0, s_1, \ldots, s_{m-1}, s_m = n]$ and a binary associative and commutative operator $\oplus$. Then it returns the array $[r_0, r_1, \ldots, r_{m-1}]$ where an element r_i is the result of reduce with $\oplus$ over the subarray $[a_{s_i}, \ldots, a_{s_{i+1}-1}]$ of A. Since a simple implementation that distributes work per subarrays is very inefficient owing to load imbalance, the implementations of Modern GPU intelligently parallelize the computations so that almost complete load balancing can be achieved.

2.2 GPU-accelerated data analysis techniques

GPUs have been successfully utilized to accelerate many data processing techniques. This section gives an overview of representative studies of databases, data mining, and graph data analyses.

2.2.1 Databases

In the database field, the following topics have been well studied using GPUs: (1) Query processing and (2) GPU-accelerated database systems.

Query processing

The main focus of GPU-accelerated query processing include sorting, joins, and indexed search. Since sorting is described in Section 2.4, the work on joins and indexed search is reviewed here.

GPU acceleration of relational joins has been studied since the introduction of CUDA. He et al. [54] discussed designs and implementations of relational-join algorithms using GPUs. They developed data-parallel primitives to harness the power of GPUs and implemented relational-join algorithms by using the primitives as building blocks. Specifically, the following four algorithms were implemented and compared: (1) non-indexed nested-loop join, (2) indexed nested-loop join, (3) sort-merge join, and (4) hash join. Their results showed that GPU implementations outperform CPU counterparts by a factor of 2 to 7. Kaldewey et al. [63] proposed an implementation of hash join by exploiting a feature of CUDA, called *unified virtual addressing (UVA)*. UVA allows GPUs to directly access the CPU main memory, thereby enabling processing larger data than the size of GPU memory. He et al. [52] accelerated hash joins by exploiting CPU–GPU integrated chips such as AMD APUs[1]. Such an architecture can remove the data-transfer time between the CPU and GPU, which often becomes the bottleneck in conventional GPU applications. They discussed and developed several kinds of fine-grained processing mechanism that is suitable for the architecture.

Acceleration of various kinds of search processing with tree structure indexes has also been studied recently. Kim et al. [66] proposed a tree index structure called *FAST (fast architecture-sensitive tree)*, suited for processing on both CPUs and GPUs. FAST is a binary tree taking into account hardware features such as the size of cache lines and page size of virtual memory, thereby not only reducing memory

[1]`http://developer.amd.com/tools-and-sdks/heterogeneous-computing/`

access latency but also making the structure suitable for parallel processing. Meanwhile, for accelerating multi-dimensional range search, an R-tree on the GPU was proposed by Kim et al. [65]. They proposed an algorithm called *MPTS (massively parallel three-phase scanning)*, to efficiently traverse an R-tree on the GPU. MPTS transforms recursive tree search into serial data processing, thereby exploiting the parallelism and resources of GPUs. Goldfarb et al. [41] noticed the pattern commonly existing in multiple tree-traversal algorithms and proposed a method to transform the tree-traversal pattern into a format that can be efficiently processed on the GPU. They proposed *autorope*, which has the following features: (1) autorope can be applied without semantic knowledge of tree-traversal algorithms; (2) autorope can be applied to algorithms with complex traversal patterns; and (3) pre-processing on tree structures is not needed.

Database systems

Database systems using GPUs have been also developed recently. He et al. [56] developed *GDB*, the first relational database systems using GPUs. They developed data-parallel primitives and implemented basic relational operations by using the primitives as building blocks. In addition, they designed a cost model to estimate query processing time on the GPU for query optimization. Heimel et al. [57] developed *Ocelot*, an extension of MonetDB[2], an open-source column-store database. Currently, various kinds of parallel processors are available, such as multi-core CPUs and GPUs. Thus it is costly to implement appropriate programs for each processor. Ocelot introduces *hardware-oblivious* operators to alleviate this challenge. These operators are implemented without awareness of specific processors, and they are compiled for the actual hardware in runtime. Thereby Ocelot reduces development costs, without sacrificing the performance. Wu et al. [135] presented *Red Fox*, a compiler and runtime infrastructure to execute relational queries on the GPU. Red Fox consists of the four components: (1) a language front-end for LogiQL [61], (2) a compiler from a relational algebra to GPU implementations, (3) optimized GPU implementations of relational operators, and (4) a supporting runtime. Red Fox is the first system that supports the complete set of TPC-H[3] queries on commodity GPUs.

[2]`https://www.monetdb.org/`

[3]`http://www.tpc.org/tpch/`

2.2.2 Data mining

In the field of data mining, GPUs have been widely utilized to enable large-scale mining. This subsection gives an overview of GPU-accelerated data mining and machine learning methods. Since association rule mining (or frequent itemset mining) and clustering, which are representative data-mining methods, are reviewed in Sections 2.3 and 2.5, respectively, the following describes GPU acceleration of other well-known techniques.

Support vector machines (SVMs) are well-known supervised learning models that are able to achieve high accuracy among other supervised learning methods. Since the learning process is time-consuming, the use of GPUs has been investigated. Catanzaro et al. [19] firstly implemented SVMs on GPUs. They parallelized the *sequential minimal optimization* (SMO) algorithm [110] on the GPU, and also employed several heuristics adaptively, thereby speeding up learning of SVMs. In addition, they accelerated not only the learning process but also classification with SVMs on the GPU. Cotter et al. [25] proposed a learning algorithm of SVMs that is more suitable for GPU processing. While existing GPU-based methods are based on parallelization of conventional algorithms, Cotter et al. designed a novel algorithm taking into account several perspectives of GPUs from scratch. An advantage of their algorithm is that it efficiently handles sparse data, which is difficult to deal with by existing methods. More recently, Li et al. [81] accelerated the learning process of SVMs that uses n-fold cross validation, which is commonly used in SVMs to tune hyper-parameters. However, it makes learning computationally expensive. Li et al. addressed this issue by parallelizing multiple learning tasks and sharing matrices of multiple tasks.

As other data mining and machine learning methods, Yan et al. [138] accelerated *LDA (latent Dirichlet allocation)* [12], a representative topic model. They parallelized on the GPU two methods used in the learning of LDA, namely *collapsed Gibbs sampling (CGS)* [45] and *collapsed variational Bayesian (CVB)* [125]. They also proposed a data-partitioning method to maximize the utilization of GPUs and a data-streaming scheme to handle larger data than the size of GPU memory. Meanwhile, Raina et al. [113] proposed GPU-accelerated two unsupervised learning methods, *deep belief networks (DBNs)* [58] and *sparse coding* [102]. To speed up learning of these methods on GPUs, the following requirements should be met: (1) to minimize data transfer between the CPU and GPU; and (2) to make the best use of hierarchical parallelism of CUDA. They firstly develop an algorithmic template that satisfies the two requirements, and on the basis of this template, the two methods are parallelized for GPUs.

Recently, *deep learning* [72] has gained tremendous attention, because it achieves high accuracy in various kinds of tasks such as speech recognition and visual object recognition. However, since the learning process is highly time-consuming, it is difficult to conduct learning in realistic time without acceleration. To address this issue, GPUs have been recently utilized, and many frameworks for deep learning using GPUs have been developed, such as cuDNN[4] published by NVIDIA, Theano[5], Caffe[6], and Chainer[7].

2.2.3 Graph data analyses

Graph data management and processing has increasingly gained much attention because of the emergence of various kinds of real-world graphs, including online social networks and biological data. Since real-world graphs may become gigantic, GPU acceleration of graph algorithms has been widely proposed. In particular, the following fundamental graph processing algorithms have been extensively explored: (1) breadth first search and (2) shortest path problems. In addition, GPU-based graph mining algorithms as well as graph processing systems have been also developed.

Fundamental algorithms

Breadth first search is a very fundamental operation to traverse graph data structures, and its GPU acceleration has been broadly studied. Harish and Narayanan [49] firstly accelerated not only breadth first search but also shortest-path problems. Luo et al. [86] pointed out the problem that the GPU implementation of breadth first search by Harish et al. has a larger computational complexity than a serial CPU implementation, and they proposed a more efficient GPU implementation. Specifically, while the computational complexity of Harish et al. is $O(nd + m)$, the method by Luo et al. has the complexity of $O(n + m)$, where n is the number of vertices, d is the diameter of a graph, and m is the number of edges, However, the method by Luo et al. has that problem that load balancing is not well considered, and the performance degrades when degree distributions follow power laws.

Hong et al. [60] presented an efficient method, called a virtual warp-centric programming method, which employs warp-oriented task assignment. In order to balance load, they introduced the following two techniques with queues: (1) deferring

[4]`https://developer.nvidia.com/cudnn`
[5]`http://deeplearning.net/software/theano/`
[6]`http://caffe.berkeleyvision.org/`
[7]`http://chainer.org/`

processing of vertices with exceptionally large degrees and (2) dynamic workload distribution. Meanwhile, Hong et al. [59] developed a parallel breadth-first-search method targeting multi-core CPUs, and combined it with the GPU implementation [60] to realize a CPU–GPU hybrid method. Their method adaptively selects the appropriate device. Specifically, if the number of traversed vertices is small at one level, the CPU is used because it incurs small overhead. On the other hand, if a large number of vertices are to be traversed, the GPU is chosen because of higher throughput.

Merrill et al. [93] proposed an efficient GPU implementation that has the optimal time complexity of $O(n + m)$ and achieves load balancing by using the scan primitive. In particular, their method exploits the scan primitive for load balancing when retrieving the adjacent vertices of current active vertices. Nasre et al. [96] compared *data-driven* and *topology-driven* implementations on irregular algorithms including graph processing. A data-driven implementation maintains a set of "active" vertices and process only such vertices. A topology-driven implementation examines all vertices and process the vertices that are considered active. Topology-driven implementations can be easily parallelizable, although the processing cost likely becomes high. On the other hand, data-driven implementations have the optimal computational complexity, while parallel implementations are difficult. Nasre et al. discussed tradeoffs between the two schemes and compare them with six applications.

To handle larger-scale graph data, Mastrostefano and Bernaschi [88] developed a method using GPU clusters. They used a load-balancing approach for a single GPU based on binary search, thereby efficiently dealing with scale-free graphs. In addition, they introduced an efficient pruning procedure to reduce communication overhead. The current state-of-the-art method is considered to be *Enterprise* proposed by Liu and Huang [83]. Their method achieves high performance of breadth first search by integrating the following three techniques: (1) *streamlined GPU threads scheduling*: constructing a frontier queue without conflicts by multiple threads; (2) *GPU workload balancing*: classifying frontier vertices on the basis of their out-degrees; and (3) *GPU-based BFS direction optimization*: caching hub vertices when switching top-down and bottom-up breadth first search [9].

As another fundamental algorithm of graph processing, efficient solutions for shortest path problems on GPUs have also been extensively studied. While there exist a number of variants of shortest path problems, two variants are commonly tackled: (1) *single-source shortest path (SSSP)* problem and (2) *all-pairs shortest path (APSP)* problem. The SSSP problem, given a weighted graph $G = (V, E, w)$ and a source vertex s, finds the minimum costs from s to all the other vertices $v \in V$.

The APSP problem, given a weighted graph $G = (V, E, w)$, finds the minimum costs of all pairs between vertices v, $u \in V$.

The pioneering work for SSSP and APSP was presented by Harish and Narayanan [49], as mentioned above. Their SSSP implementation is based on parallelization of Dijkstra's algorithm [24]. Later, Martín et al. [87] pointed out that the implementation of SSSP by Harish et al. computes results different from the optimal one, and presented a solution to this issue. The implementation of Harish et al. has the problem that concurrency control is not perfect. Thus Martín et al. proposed solutions that compute the optimal result. Delling et al. [29] developed a pre-processing-based SSSP method, called *PHAST*, especially focusing on certain classes of graphs, including road networks. PHAST constructs an index structure from a graph on the basis of *contraction hierarchies* [40]. This index enables the SSSP computation with only a linear sweep of all vertices. PHAST also takes advantages of parallelism available on multi-core CPUs and GPUs. More recently, Davidson et al. [27] presented three methods for SSSP, *workfront sweep*, *near-far pile*, and *bucketing*, by balancing the tradeoff between saving work and organizational overhead. These methods are designed to expose sufficient parallelism for GPUs. They reported that near-far pile is the fastest method because of its balance between work efficiency and available parallelism.

As for the APSP problem, Katz and Kider [64] proposed a method based on the Floyd–Warshall algorithm [24]. Their method partitions the adjacency matrix into small blocks and processes each block on shared memory, thereby accelerating memory accesses to graph data. They also developed methods to handle larger data than GPU memory and also to utilize multiple GPUs. Buluç et al. [16] presented an APSP method that recursively partitions the APSP computations and exploits existing implementations of fast matrix multiplication. Meanwhile, Matsumoto et al. [89] proposed a similar method in that they recursively partition the APSP computations and utilize matrix multiplications. The difference is that the method of Matsumoto et al. computes not only the minimum costs of shortest paths but also the actual shortest paths, and their method is able to cope with large-scale data that does not fit into GPU memory. Okuyama et al. [101] developed an improved method of the APSP implementation by Harish and Narayanan [49]. The implementation by Harish et al. solves the APSP problem by separately computing SSSP from all vertices as sources. On the other hand, Okuyama et al. exploit the property that intermediate results of multiple SSSP problems can be shared, by using shared memory.

The above-mentioned APSP methods deal with only small-scale graphs, at most 32,000 vertices. To solve larger-scale problems, Djidjev et al. [30] proposed a scheme for handling huge graphs using GPU clusters. Their scheme distributes the APSP computation by partitioning the input graph. Specifically, their method realizes distributed APSP computations by the following four steps: (1) the input graph is divided into *k* components; (2) each component is processed by the Floyd–Warshall algorithm on a GPU; (3) the method is recursively applied to boundary vertices in each component; and (4) with the results of steps 2 and 3, shortest paths crossing different components are updated.

Other techniques

Data mining methods targeting graph data have also been developed, including PageRank [105], HITS [67], and graph clustering [35]. Since graph clustering is described in Section 2.6, other methods are reviewed here.

Yang et al. [140] proposed a fast method of sparse matrix vector multiplication (SpMV), thereby accelerating graph mining algorithms such as PageRank and HITS, because these algorithms are mostly based on SpMV. Their method is designed to capture the characteristics of real-world graphs, especially the scale-free property [7]. They also discussed a scheme to automatically tune parameters used in their method.

He et al. [55] accelerated SimRank [62], a similarity measure between vertices in a graph. While existing methods of SimRank have two limitations that they are computationally expensive and only applicable to static graphs, They proposed a novel technique to rewrite the SimRank equation so that optimizations and incremental updates can be performed. In addition, they develop a general framework for parallel SimRank computation on GPUs.

McLaughlin and Bader [91] sped up the computation of *betweenness centrality* [37], which is a popular metric to distinguish influential vertices in a graph. Existing GPU implementations manage scale-free graphs, but their performance degrades when handling graphs with large diameters such as road networks. McLaughlin and Bader alleviated this issue by developing a hybrid method. This method adaptively selects an appropriate processing scheme according to the size of a next working set.

Graph processing systems

As described above, many graph processing algorithms have been successfully accelerated by using GPUs. However, development of efficient programs for GPUs is still difficult and costly. To address this challenge, graph processing systems using

GPUs have recently been proposed.

Zhong and He [142] presented a programming framework, *Medusa*. Medusa enables users to develop programs of GPU-accelerated graph processing by writing C/C++ codes of serial execution. To efficiently process graph data on GPUs, they propose a programming model called *edge-message-vertex (EMV)*, and Medusa offers a small set of APIs based on EMV. They also introduced a series of graph-centric optimizations for GPUs and extend Medusa to execute on multiple GPUs. Fu et al. [38] also proposed a programming framework called *MapGraph*, which offers APIs based on the modified *gather-apply-scatter (GAS)* model [42]. MapGraph achieves high performance by dynamically selecting scheduling strategies depending on the frontier size and the size of adjacency lists of frontier vertices. They also integrated several existing optimizations suitable for GPU architectures.

More recently, Wang et al. [130] developed a graph processing system, *Gunrock*. Gunrock simplifies graph processing on the GPU by a high-level bulk-synchronous abstraction with traversal and computation steps. They also integrated a number of optimization strategies such as multiple load-balancing strategies, direction-optimal traversal, and a two-level priority queue.

2.3 Frequent itemset mining

The *association rule mining* problem was firstly introduced by Agrawal et al. [3]. *Frequent itemset mining* can be considered as the first step for association rule mining, and usually this first step is the most time-consuming part. To accelerate this step, many efficient algorithms have been developed. Among such algorithms, there are two major algorithms, namely *Apriori* [5] and *FP-growth* [47].

Parallelization of these algorithms has been widely studied in the literature [4, 82, 84, 107, 117]. Three Apriori-based algorithms on a shared-nothing architecture were presented by Agrawal and Shafer [4]. They explored several tradeoffs such as computation, communication, and synchronization. Parthasarathy et al. [107] parallelized the Apriori algorithm that uses hash trees. They developed a parallel implementation on a shared-memory multi-processor, taking into account load balancing, data locality, and false sharing. Liu et al. [84] proposed a cache-conscious FP-array to improve the spatial locality of FP-growth, and a lock-free dataset tiling approach to fully utilize a modern multi-core processor. Li et al. [82] designed a parallel algorithm of FP-growth on a massive computing environment using MapReduce [28]. More recently, mcEclat was introduced by Schlegel et al. [117] to handle the large number of threads provided by Intel many-core processors.

While the above-mentioned parallel algorithms work well for the conventional *certain* transaction databases, they cannot effectively process frequent itemset mining from uncertain databases, which gains increasing importance in order to handle data uncertainty. There is much existing work for modeling, querying, and mining such uncertain data (see a survey by Aggarwal and Yu [2] for details and more information).

To mine frequent itemsets with taking into account the uncertainty, a large number of algorithms have been proposed. Chui et al. [22] proposed the *U-Apriori* algorithm that computes the expected support of itemsets by summing up all itemset probabilities under the attribute-uncertainty model, where each item in transactions has an existential probability. Leung et al. [77] proposed a tree-based mining algorithm *UF-growth*, which is an adaptation of FP-growth. Later, Leung and Tanbeer improved the algorithm and compactness of the tree structure [75]. Sampling-based method was also introduced by Calders et al. [18].

Bernecker et al. [11] found that the use of expected supports may result in missing interesting itemsets, and proposed an algorithm to find probabilistic frequent itemsets. While all the above-mentioned work deals with the attribute-uncertainty model, Sun et al. [124] considered the tuple-uncertainty model, where each transaction has an existential probability, and they presented algorithms for mining probabilistic frequent itemsets under this model. Recently, Tong et al. [128] implemented existing representative algorithms and test their performance with uniform measures fairly.

Several attempts to accelerate the algorithms also exist [17, 76, 132]. Calders et al. [17] sped up the computation by approximating the probabilities for probabilistic frequent itemsets based on sampling. Wang et al. [132] accelerated the computation of probabilities by exploiting the statistical properties of probabilistic models. Leung and Hayduk [76] proposed a tree-based algorithm with MapReduce to find expected support-based frequent itemsets. To the best of our knowledge, there has been no work to accelerate probabilistic frequent itemset mining with GPUs.

Frequent itemset mining from "certain" databases on GPUs was studied in [6, 34, 120, 127]. Fang et al. [34] proposed two approaches, a GPU-based method and a CPU–GPU hybrid method. The GPU-based method uses bitstrings and bitwise operations for fast support counting, running entirely on the GPU. The hybrid method adopts the trie structure on the CPU and counts supports on the GPU. Teodoro et al. [127] parallelized the Tree Projection algorithm [1] on the GPU as well as multicore CPU. Amossen and Pagh [6] presented a novel data layout *BatMap* to represent bitstrings, which is well suited to parallel processing and compact even for sparse data. They make use of BatMap to accelerate frequent itemset mining. Silvestri and

Orlando [120] proposed a parallel version of a state-of-the-art algorithm *DCI* [103].

2.4 Sorting

Sorting is a fundamental operation in many algorithms, and its acceleration has significant importance in many fields. Recently, GPU acceleration of sorting has also been extensively studied.

The pioneering work was done by Govindaraju et al. [43] before the introduction of CUDA. They presented a novel sorting algorithm called *GPUTeraSort*, which has the following advantages: (1) it can sort datasets that do not fit into the GPU memory; and (2) it can handle long records with wide keys.

Since the introduction of CUDA, a large number of proposals have been made. Satish et al. [115] discussed designs and implementations of radix sort and merge sort on GPUs. An efficient implementation of radix sort is realized by exploiting the scan primitive for enabling fine-grained parallel processing. Merge sort is implemented by developing an efficient algorithm for merging a pair of sorted sequences. In addition, they also discussed the use of shared memory for the sorting algorithms. A faster comparison-sorting algorithm[8] was proposed by Ye et al. [141], called *GPU-Warpsort*. GPU-Warpsort exploits the lock-step nature of CUDA's warps, thereby reducing the overhead of thread synchronization.

As another comparison-sorting algorithm, Leischner et al. [73] presented a GPU implementation of samplesort [13]. Samplesort is an algorithm that recursively partitions an input sequence into k buckets by sampling $k - 1$ splitters from the input. When the input becomes sufficiently small, another algorithm is used to finish the sorting. Compared to merge sort and quicksort, samplesort has the advantage that the number of global-memory accesses is smaller. Samplesort is described in Section 4.2.1 in more detail.

Meanwhile, Satish et al. [116] discussed efficient implementations of merge sort and radix sort, targeting both CPUs and GPUs. For a fair comparison, they tune implementations for both architectures. As a result, the CPU radix sort outperforms the GPU counterpart by 20%. On the other hand, the merge sort on the GPU shows higher performance that on the CPU.

The state-of-the-art implementation of radix sort is considered to be the work presented by Merrill and Grimshaw [92]. They exploit two techniques, *kernel fu-*

[8]Comparison sorting algorithms here mean that the algorithms sort input lists by only comparing pairs of elements [68]. For example, merge sort is a comparison-sorting algorithm, while radix sort is not.

sion [136] and *multi-scan*, thereby largely reducing memory-access cost, compared with existing methods. Kernel fusion attempts to combine multiple kernels into one, resulting in better utilization of GPU resources. Multi-scan computes multiple scan operations simultaneously, reducing the computational cost. On the basis of this implementation, a more tuned one is publicly available as a part of library called *CUB* [94].

On the other hand, the state of the art of comparison sorting is a merge-sort-based method published as a part of library, *Modern GPU* [8]. This method is based on *merge path* [44], which partitions merge of a pair of sorted lists into p equi-sized subproblems. Thus, by using merge path, completely load-balanced merge can be achieved. The Modern GPU implementation utilizes such merge procedures to realize an efficient merge sort.

2.5 Clustering

Clustering is a task to separate data points from a dataset such that similar points belong to the same group called *cluster* [48]. In general, standard clustering algorithms require high computational cost. For example, k-means [48] has the time complexity of $O(nkt)$, where n is the number of data points, k is the number of clusters, and t is the number of iterations. As another example hierarchical agglomerative clustering [48] has the time complexity of $O(n^3)$ with n data points. Thus there are a large number of techniques to accelerate the processing of clustering algorithms. One of such techniques is *canopy clustering* [90], which is described in Section 5.2 in detail. Another option for acceleration is to parallelize the computation of clustering algorithms. The following first reviews parallel clustering on CPUs and then describes clustering on GPUs.

2.5.1 Parallel clustering

Significant efforts have been made to parallelize clustering algorithms. For instance, Dash et al. [26] proposed a parallel algorithm of hierarchical agglomerative clustering for shared-memory architectures that employs partially overlapping partitioning. Patwary et al. [108] parallelized the DBSCAN algorithm [48] by using the disjoint-set data structure and a tree-based bottom-up scheme to build clusters. More recently, Li et al. [79] presented an approach of k-means for the MapReduce framework. They utilize three techniques: locality sensitive hashing, a novel center initialization algorithm, and a pruning scheme for avoiding unnecessary distance computations. To the best of our knowledge, only Soroush et al. [122] dealt with the parallelization of canopy clustering. They use canopy clustering as a running example for their new storage manager for parallel array processing, *ArrayStore*, and implement canopy clustering on top of ArrayStore.

2.5.2 GPU clustering

Clustering on GPUs has been extensively studied since the introduction of CUDA [98]. In particular, the k-means algorithm is well studied because of its popularity [69, 80, 133]. Wasif and Narayanan [133] implemented all the steps of k-means entirely on the GPU, and evaluated the implementation not only on a single GPU but also on multi-GPU platforms. Kohlhoff et al. [69] also presented an efficient implementation of k-means for GPUs by utilizing parallel sorting as preprocessing. Another implementation of k-means for GPUs is proposed by Li et al. [80]. They take into account the dimensionality and adaptively employ two schemes for low-dimensional data and high-dimensional data, respectively.

There also exists work on the acceleration of other algorithms such as DBSCAN and hierarchical agglomerative clustering [15, 118, 134]. Böhm et al. [15] proposed *CUDA-DClust*, a parallel algorithm for density-based clustering on GPUs. They further accelerated this algorithm by using an index structure suited for GPU processing. Another variant of DBSCAN, called Mr. Scan, is introduced by Welton et al. [134]. This algorithm is implemented on the MRNet tree-based distribution network with GPU-configured nodes, and is capable of clustering 6.5 billion points in 17.3 minutes. Hierarchical agglomerative clustering was accelerated Shalom and Dash [118]. They use partially overlapping partitions in order to efficiently parallelize the computation on GPUs.

2.6 Graph clustering

Graph clustering, also known as *community detection*, has recently been paid significant attention, and many algorithms and measures to evaluate clustering results have been proposed [35, 131]. This section gives an overview of graph clustering algorithms (Section 2.6.1) and their parallelization (Section 2.6.2).

2.6.1 Serial graph clustering

Modularity [97] is a popular measure to evaluate clustering quality, and many modularity-based algorithms have been proposed. Modularity measures the difference between a clustering result and an expected random graph, and it takes high values when the result contains many intra-cluster edges and few inter-cluster edges. The *Louvain* method [14] is a well-known greedy algorithm to maximize modularity. Louvain enables fast clustering by computing modularity gains when a vertex is moved from a cluster to another. Shiokawa et al. [119] improved the Louvain method by incrementally aggregating clusters. However, it is pointed out that modularity has a limitation called *resolution limit* [36], which means that, as graphs become larger, small clusters cannot be captured by modularity.

As another class of graph clustering algorithms, *label propagation* has been proposed [78, 112]. Raghavan et al. [112] proposed a basic label propagation algorithm, which clusters graph data by propagating labels from vertices to neighbors. Leung et al. [78] improved the basic label propagation algorithm by introducing two heuristics, called *hop attenuation* and *node preference*.

Wang et al. [131] conducted extensive experiments for fair comparisons of ten graph clustering algorithms, by implementing them with a common framework. Consequently, label propagation achieves higher performance and more accurate results compared with other algorithms. Therefore, this study focuses on the acceleration of label propagation.

2.6.2 Parallel graph clustering

Since real-world graphs become increasingly huge, it is important to accelerate graph clustering techniques. To this end, many parallel graph clustering methods have recently been developed. In particular, because of its popularity, parallelization of modularity-based algorithms has been extensively studied. For instance, Djidjev and Onus [31] accelerated modularity-based clustering by utilizing the property that the problem of modularity maximization can be reduced to a minimum weighted cut problem on a complete graph with the same vertices. With this reduction, modularity maximization can be efficiently solved by existing parallel graph partitioning techniques. Que et al. [111] parallelized the Louvain method on a large-scale cluster consisting of 8,192 nodes. They introduce a novel hash-based strategy to efficiently store and process dynamic graphs.

Parallelization of label propagation has also been proposed, because it is suitable for parallel processing as only vertex-local information is used. Cordasco and

Gargano [23] proposed a novel method that updates vertex labels *semi-synchronously*. In label propagation algorithms, synchronous label updates mean that all labels are updated at the same time, and asynchronous label updates mean that labels are sequentially updated [112]. The synchronous updates can be easily parallelizable, but it is known that the convergence is slow [78]. On the other hand, the asynchronous updates converge faster, but it is difficult to correctly parallelize them. The semi-synchronous method is a hybrid of synchronous and asynchronous methods: it partitions a graph into multiple sets by graph coloring [], labels are updated in a set-by-set fashion, and within each set, labels are updated synchronously. Although Cordasco et al. did not conduct experiments with parallel environments, later Duriakova et al. [32] extended and implemented the semi-synchronous scheme and evaluated it with multi-core CPUs.

GPUs have also been utilized to accelerate graph clustering. Stovall et al. [123] parallelized on GPUs the *SCAN* algorithm [137], which is based on structural similarity. Label propagation on GPUs was proposed by Soman and Narang [121]. They develop a parallel label propagation method targeting both multi-core CPUs and GPUs. However, the details of GPU implementation is not well discussed and load balancing is not considered much.

2.7 Discussion

This chapter has given an overview of related work on GPU-accelerated data analysis techniques and the four problems: frequent itemset mining, sorting, clustering, and graph clustering. This dissertation presents GPU-accelerated methods for the problems.

A GPU-accelerated method of frequent itemset mining from uncertain data is described in Chapter 3. While GPU acceleration of frequent itemset mining from "certain" data has already been proposed [6, 34, 120, 127], our target is to speed up frequent itemset mining over uncertain data, which has not yet been accelerated by GPUs. More than that, the key component is the acceleration of numerous convolutions, and this kind of computation has not been accelerated by GPUs, to the best of our knowledge.

A novel algorithm of efficient comparison sorting on GPUs is provided in Chapter 4. Since comparison sort can be more widely applicable than non-comparison sort such radix sort, we attempt to accelerate comparison sorting on GPUs. The state of the art of comparison sorting is a merge-sort-based implementation presented by Baxter [8]. However, it has the limitation that the performance degrades for sorting large-scale data. Our method alleviates this limitation by combining Baxter's implementation and samplesort [73], with taking into account load balancing.

Chapter 5 presents efficient algorithms and implementations of canopy clustering on GPUs. While standard clustering algorithms such as k-means have been extensively accelerated by GPUs, canopy clustering on GPUs has not yet been tackled. With the acceleration of canopy clustering, we can expect to speed up the whole clustering process, by combining GPU-accelerated canopy clustering and clustering algorithms. More than that, we also propose an efficient procedure to construct grid index structures on the GPU by exploiting data-parallel primitives.

GPU-accelerated graph clustering is introduced in Chapter 6. Our method is based on parallelization of label propagation [112], one of the fastest algorithms. Label propagation is suitable for parallel processing [78] and achieves accurate clustering results [131]. Thus label propagation is considered as a promising algorithm for parallelizing on GPUs. Soman and Narang [121] has already presented a parallel label propagation algorithm for GPUs as well as multi-core CPUs. However, their method does not well take into account load balancing, which is extremely important to efficiently handle real-world graphs. On the other hand, our method achieves load balancing almost completely, by exploiting primitives whose performance is not largely affected by the existence of skewness.

While this dissertation focuses on the use of GPUs, another option of many-core processors is available: Intel® Xeon Phi™. Intel® Xeon Phi™ includes around 60 cores, each of which supports long (512-bit) SIMD instructions, thereby enabling massive and hierarchical parallelism. As for threading models, GPUs employ massive amounts of light-weight threads that can be swithched with almost zero overhead. On the other hand, Intel® Xeon Phi™ utilizes threads with costly context switches, and each core accommodates up to four threads. Therefore, compared to Intel® Xeon Phi™, GPUs excel in computing massive amounts of simple and independent computations. This means that GPUs are suitable for highly data-parallel processing. Many data analysis techniques can be efficiently processed in a highly data-parallel manner. Therefore, GPUs are considered as a more promising solution than Intel® Xeon Phi™ for accelerating large-scale data analysis techniques. In particular, the four problems focused in this dissertation contain the following data-parallel nature: (1) frequent itemset mining from uncertain data requires massive amounts of probability computations; (2) comparison sorting can be efficiently expressed as a set of data-parallel primitives; (3) canopy clustering needs many distance computations between data points; and (4) graph clustering has data-parallelism in the levels of vertices and edges.

Actually, a number of research papers have reported that GPUs exhibit better performance and more effective memory bandwidth than Intel® Xeon Phi™ for various applications [20, 126, 129]. Teodoro et al. [126] conducted comparative performance analyses of Intel® Xeon Phi™, GPUs, and CPUs with a case study of microscopy image analysis. Applications of microscopy image analysis consist of multiple common operations. Teodoro et al. identify the data access and computational patterns of the operations, and they classify the operations into three categories: regular data access, irregular data access, and heavy use of atomic functions. Their experimental results reveal the following observations: (1) GPUs are faster on memory-bound operations with regular data access; (2) GPUs and Intel® Xeon Phi™ are comparable on compute-bound operations in the category of regular data access; (3) GPUs are more efficient than the other devices on operations of irregular data access; and (4) GPUs significantly outperforms the other devices on operations that heavily rely on atomic functions. Tran et al. [129] investigated acceleration of bit-parallel approximate pattern matching on both GPUs and Intel® Xeon Phi™. Their results also show that a GPU implementation achieves superior performance than Intel® Xeon Phi™ by a factor of up to 2.9. By observing these facts, GPUs are considered as a more promising solution for accelerating large-scale data analysis techniques.

Chapter 3

GPU-Accelerated Frequent Itemset Mining from Uncertain Data

Uncertainty is prevalent in many real-world applications such as sensor monitoring systems [2]. To deal with the vast amount of uncertain data, uncertain databases have been widely studied during the last decade. Several algorithms of frequent itemset mining, one of the major data mining techniques, have also been proposed to extract valuable information from uncertain databases [128]. However, their performance is not satisfactory because handling uncertainty incurs high processing cost. In order to address this problem, we utilize GPUs, which increasingly gain popularity as many-core processors. The key component is to speed up probability computations by making the best use of GPU's high parallelism and fast local memory. More specifically, we accelerate numerous convolutions between probability mass functions by exploiting their tree-structured computation. In addition, we present a technique of simultaneous computation of convolutions to maximize the utilization of GPUs. We also employ an algorithm to manipulate bitstrings and data-parallel primitives to improve performance in other steps. Extensive experiments show that our GPU implementation is up to 5.5 times faster than a parallelized CPU implementation.

3.1 Introduction

The problem of uncertain data management has attracted a great deal of attention due to the inherent uncertainty in the real world [2]. For example, when analyzing purchasing behavior of customers using RFID sensors, there may be incorrect sensor readings due to errors. In addition to this, uncertain data takes place in many situations. Major sources of uncertainty include limitations of equipment, privacy concerns, or statistical methods such as forecasting. To deal with such uncertain data, *uncertain databases* have recently been developed [2].

In the area of uncertain data management, frequent itemset mining or association rule mining [3] from uncertain databases is one of the major research issues, and there have been a number of algorithms [11,17,18,22,75,77,124,128,132]. Although conventional frequent itemset mining is a well-studied technique [3,5,47], frequent itemset mining from uncertain databases is different from the conventional one in the sense that we need to take into account uncertainty. Existing algorithms suffer from performance problems due to the fact that dealing with uncertainty, such as probability computation, incurs extra cost and is often highly time-consuming. It is thus necessary to accelerate this computation in order to handle large-scale uncertain databases.

This chapter presents a method of GPU-accelerated frequent itemset mining from uncertain databases. The key component is to speed up probability computations by making the best use of GPU's high parallelism and fast local memory. More specifically, we accelerate numerous convolutions between probability mass functions by exploiting their tree-structured computation. In addition, we present a technique of simultaneous computation of convolutions to maximize the utilization of GPUs. We also employ an algorithm to manipulate bitstrings and data-parallel primitives to improve performance in other steps. Extensive experiments show that our GPU implementation is up to 5.5 times faster than a parallelized CPU implementation.

3.2 Preliminaries

Section 3.2.1 defines conventional frequent itemsets, and Section 3.2.2 extends the definition for uncertain data. This study basically follow the definition provided by Sun et al. [124]. Section 3.2.3 describes the pApriori algorithm, which was proposed by Sun et al. [124].

3.2.1 Frequent itemsets

Let I denote a set of all items. A set $X \subseteq I$ of items is called an *itemset* and a k-itemset means an itemset that contains k items. A pair of an ID and an itemset is called a *transaction* T.[1] A *transaction database* $\mathcal{T}$ is a set of transactions and its cardinality is denoted by $|\mathcal{T}|$. Given a transaction database $\mathcal{T}$ and an itemset X, the *support* of X, denoted by $\text{sup}(X)$, is defined as the number of transactions in $\mathcal{T}$ that include X. An itemset X is *frequent* or a *frequent itemset* if $\text{sup}(X) \geq \texttt{minsup}$, where `minsup` is a support threshold given by users.

3.2.2 Probabilistic frequent itemsets

An *uncertain transaction database* $\mathcal{U}$ is a transaction database where each transaction has an *existential probability*. The probability stands for the chance that a transaction exists in $\mathcal{U}$. Table 3.1 shows an uncertain transaction database, which represents purchase records. For example, the transaction T_1 means that items a and b are purchased together with probability 0.8.

Uncertain transaction databases are often interpreted with the possible world semantics [124, 132]. Under this notion, a database generates a set of *possible worlds*, each of which is a distinct combination of transactions in the database. For instance, the database in Table 3.1 produces 16 possible worlds: $\emptyset, \{T_1\}, \{T_2\}, \{T_3\}, \{T_4\}, \{T_1, T_2\}$, etc. Each world is also associated with an existential probability that is the product of the probabilities of world's transactions. The world $\{T_1\}$ in Table 3.1 has the probability that the transaction T_1 exists and the transactions T_2, T_3, and T_4 do not exist (i.e., $0.8 \cdot (1 - 0.7) \cdot (1 - 0.9) \cdot (1 - 0.5) = 0.012$).

In this interpretation, the support of an itemset X varies with possible worlds and hence becomes a random variable. The probability mass function of a support is called a *support probability mass function (SPMF)*, denoted as f_X. With this function, the probability that $\text{sup}(X) = i$ can be written as $f_X(i)$, where i is an integer that $\text{sup}(X)$ may take (i.e., $i \in \{0, 1, ..., |\mathcal{U}|\}$). In principle, $f_X(i)$ is calculated by summing the probabilities of worlds where $\text{sup}(X) = i$. For example, Figure 3.1 illustrates an SPMF $f_{\{\text{b}\}}$ for the database in Table 3.1. The support $\text{sup}(\{b\})$ equals 0 in worlds $\emptyset$ and $\{T_3\}$. Hence $f_{\{b\}}(0)$ is evaluated as $0.003 + 0.027 = 0.03$, where "0.003" and "0.027" are the existential probabilities of worlds $\emptyset$ and $\{T_3\}$, respectively.

An itemset X is *probabilistic frequent* or a *probabilistic frequent itemset (PFI)* if

$$P\left(\text{sup}(X) \geq \texttt{minsup}\right) = \sum_{i=\texttt{minsup}}^{|\mathcal{U}|} f_X(i) \geq \texttt{minprob}, \tag{3.1}$$

where `minprob` $\in (0, 1]$ is a probability threshold given by users. In short, an

[1] A *transaction* is used to mean the *itemset* of the transaction in the context of set-theoretic inclusion relation.

Table 3.1: An uncertain transaction database.

ID	Itemset	Prob.
T_1	{a, b}	0.8
T_2	{b, c}	0.7
T_3	{a}	0.9
T_4	{a, b, c}	0.5

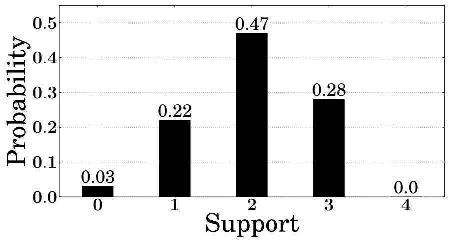


Figure 3.1: The SPMF of an itemset {b}.

itemset is probabilistic frequent if it is frequent with certainty at least `minprob`. For example, when `minsup` = 2 and `minprob` = 0.5 with the database in Table 3.1, an itemset {b} is a PFI because

$$\begin{aligned} &P(\mathrm{sup}(\{\mathrm{b}\}) \geq \texttt{minsup}) \\ &\quad = f_{\{\mathrm{b}\}}(2) + f_{\{\mathrm{b}\}}(3) + f_{\{\mathrm{b}\}}(4) \\ &\quad = 0.47 + 0.28 + 0.0 = 0.75 \geq \texttt{minprob} \end{aligned}$$

Having defined uncertain transaction databases and probabilistic frequent itemsets, we define the problem accelerated in this chapter as follows.
Problem 1 (probabilistic frequent itemset mining)**.** *Given an uncertain transaction database $\mathcal{U}$,* `minsup`*, and* `minprob`*, find probabilistic frequent itemsets from the database $\mathcal{U}$.*

3.2.3 The pApriori algorithm

The Apriori algorithm, which was originally proposed by Agrawal and Srikant [5], is a well-known algorithm for association rule mining. Sun et al. [124] proposed *pApriori* that adapts Apriori to uncertain transaction databases.

A pseudo-code of the pApriori algorithm is shown in Algorithm 1. First of all, this algorithm extracts a set C_1 of *candidate* 1-itemsets from an input database $\mathcal{U}$. This set contains an itemset whose element is one item in the database $\mathcal{U}$. From the candidates C_1, pApriori computes a set $\mathcal{L}_1$ of probabilistic frequent 1-itemsets (or 1-PFIs). Subsequently, the algorithm sets $k = 1$ and continues the following procedures alternately with incrementing k by one, until no additional PFI is detected:

1. Generating a set C_k of candidate k-itemsets from a set $\mathcal{L}_{k-1}$ of $(k-1)$-PFIs.

2. Extracting a set $\mathcal{L}_k$ of k-PFIs from the set C_k.

Algorithm 1: The pApriori algorithm.

```
   Input: uncertain transaction database $\mathcal{U}$,
          minsup, minprob
1  $C_1 \leftarrow$ generate candidates from the database $\mathcal{U}$
2  $\mathcal{L}_1 \leftarrow$ extract probabilistic frequent itemsets from $C_1$
3  $k \leftarrow 1$
4  while $\mathcal{L}_k \neq \emptyset$ do
5      $k \leftarrow k + 1$
6      $C_k \leftarrow$ generate candidates from $\mathcal{L}_{k-1}$
7      $\mathcal{L}_k \leftarrow$ extract probabilistic frequent itemsets from $C_k$
8  end
9  return All probabilistic frequent itemsets $\bigcup_{i=1}^{k-1} \mathcal{L}_i$
```

Eventually the pApriori algorithm returns all PFIs extracted from $\mathcal{U}$.

The following sections describe the two procedures and a pruning technique.

Generating candidates

The procedure of generating candidate itemsets can be separated into two phases: merging and pruning phases.

In the merging phase, candidate k-itemsets are generated from $(k-1)$-PFIs. More specifically, a candidate k-itemset is created from a pair $\{X, Y\}$ of $(k-1)$-PFIs if the pair is *joinable*, which means that X and Y satisfy the following condition: $(X[1] = Y[1]) \wedge (X[2] = Y[2]) \wedge \cdots \wedge (X[k-2] = Y[k-2]) \wedge (X[k-1] < Y[k-1])$, where $X[i]$ denotes the ith item of X in a certain order such as lexicographical order. If X and Y are joinable, a candidate k-itemset is formed as the union of X and Y, and it is stored into a set C_k of candidate k-itemsets.

In the pruning phase, the following lemma is used in order to prune the candidates [124].

Lemma 1 (Anti-monotonicity)**.** *If an itemset X is a PFI, then any itemset $X' \subset X$ is also a PFI.*

The contraposition of this lemma yields that if any itemset $X' \subset X$ is not a PFI, then the itemset X is not a PFI either. Hence a candidate k-itemset can be pruned out when any $(k-1)$-subset of the candidate is not included in a set $\mathcal{L}_{k-1}$ of $(k-1)$-PFIs. If a candidate can be pruned out, it is deleted from C_k.

Extracting probabilistic frequent itemsets

In order to reveal whether or not an itemset X is a PFI, the SPMF of X needs to be computed and substituted for Equation 3.1. Since the number of possible worlds increases exponentially with the number of transactions, a naïve solution to compute SPMFs is considered to be intractable. To address this problem, Sun et al. proposed two algorithms based on dynamic programming and divide-and-conquer [124]. Here, we only describe the latter algorithm because it is shown to be more efficient and is considered to be more suitable for parallel processing.

The algorithm takes as inputs an uncertain transaction database $\mathcal{U}$ and an itemset X and returns the SPMF f_X of X. If the database $\mathcal{U}$ contains only one transaction, it just computes the SPMF and returns the result. If $\mathcal{U}$ contains two or more transactions, it horizontally partitions $\mathcal{U}$ into two databases $\mathcal{U}_1$ and $\mathcal{U}_2$, and recursively applies the same algorithm to them. Having finished the computation for $\mathcal{U}_1$ and $\mathcal{U}_2$, the algorithm combines the resulting SPMFs into one SPMF and returns it as output.

The key here is that two SPMFs are efficiently combined into one by convolution [124]; i.e., the desired SPMF f_X can be computed as

$$f_X(i) = \sum_{j=0}^{i} f_X^1(j) \cdot f_X^2(i-j) \quad (i \in \{0, 1, ..., |\mathcal{U}|\}),$$

where f_X^1 and f_X^2 are SPMFs on $\mathcal{U}_1$ and $\mathcal{U}_2$, respectively. Although the naïve computation of convolution requires $O(|\mathcal{U}|^2)$ time, it can be improved to $O(|\mathcal{U}| \log(|\mathcal{U}|))$ time complexity with fast Fourier transform (FFT) algorithms [124]. Consequently, the time complexity of the divide-and-conquer algorithm results in $O(|\mathcal{U}| \log^2(|\mathcal{U}|))$.

Pruning

Owing to the high complexity of SPMF computation, it is desirable to prune infrequent itemsets without computing SPMFs. For this purpose, a count and an expected support of an itemset X are used. The *count* $\mathrm{cnt}(X)$ is the number of transactions that include an itemset X, regardless of existential probabilities. The *expected support* $\mathrm{esup}(X)$ is the expectation of support $\sup(X)$, which can be computed by summing the existential probabilities of transactions that include X. Sun et al. proved two lemmas that allow for pruning candidates with the two values [124]. Since counts and expected supports can be obtained by a single scan of database, it is more efficient to compute counts and expected supports than to compute SPMFs.

(1)

0	1	X
1	2	X
0	X	X
0	1	X

(2)

0.8
0.7
0.9
0.5

Figure 3.2: An uncertain transaction database on the GPU.

3.3 Proposed method

Although the above-mentioned pApriori algorithm achieves an efficient way of probabilistic frequent itemset mining, even better performance is desirable when dealing with huge databases. To this end, we aim at improving performance of the pApriori algorithm by exploiting the massive parallelism of GPUs.

We now describe a method of probabilistic frequent itemset mining using a single GPU. We assume in the following that items in transactions are denoted as integer values in $\{0, 1, ..., |I| - 1\}$. In Section 3.3.1, we explain data layouts on the GPU. Sections 3.3.2 and 3.3.3 describe algorithms for generating candidate itemsets and extracting PFIs, respectively.

3.3.1 Data layouts on the GPU

An uncertain transaction database is the only data needed to execute our method. On the GPU, a database $\mathcal{U}$ is represented by two one-dimensional arrays: an array of itemsets in transactions and an array of existential probabilities. The first array can be regarded as a binary matrix where the entry in the ith row and jth column is 1 if the ith transaction contains the item j. Since this matrix tends to be sparse, we store the matrix using a sparse matrix format. More specifically, we employ the ELL format [114], because it is known that this format is well-suited to the GPU architecture [10]. The ELL format stores an M-by-N sparse matrix with at most K nonzeros per row as two arrays: a dense M-by-K array of nonzeros and an M-by-K array of nonzero-column indexes. When considering itemsets of transactions as a matrix, each entry contains merely 0 or 1. Thus we can represent the itemsets by storing only the latter array. Note that K is the maximum size of transactions in our case.

For example, the database in Table 3.1 can be represented as arrays shown in Figure 3.2, where the items a, b, and c are denoted as 0, 1, and 2, respectively. Each row of the first array represents the itemset of a transaction. Since the length of rows is the maximum size of transactions, shorter rows are padded with a sentinel denoted

Algorithm 2: Extract probabilistic frequent itemsets.

```
1  for $X \in C_k$ in parallel do
2      Inclusion check of $X$
3  end
4  for $X \in C_k$ in parallel do
5      Attempt to prune $X$
6  end
7  for each $X \in C_k$ do
8      Filter transactions
9      Compute the SPMF of $X$
10 end
```

as X. We used the number of items $|I|$ as the sentinel in our implementation. The second array just stores existential probabilities corresponding to the transactions.

The first array in Figure 3.2 is stored in column-major order. This means that the array is laid out in linear memory as 0, 1, 0, 0, 1, 2, X, 1, X, and so on. This layout enables GPUs to coalesce multiple memory accesses into one transaction when processing inclusion check, which is described in Section 3.3.3.

Meanwhile, we use an array of integers to represent itemsets. A k-itemset is stored in an integer array of k elements. Let now $\mathcal{S}_k$ be a set of k-itemsets, which corresponds to a set of either k-PFIs or candidate k-itemsets. The set $\mathcal{S}_k$ is stored in an integer array of $k \cdot |\mathcal{S}_k|$ elements in the following way. The first itemset in $\mathcal{S}_k$ is stored in the array from 0 to $k-1$ elements, and the second itemset is stored in the array from k to $2 \cdot k - 1$ elements. In other words, the ith itemset occupies the elements from $(i-1) \cdot k$ to $i \cdot k - 1$.

3.3.2 Generating candidates

The algorithm to generate candidates is similar to that of pApriori, while the details are different to adapt to GPUs.

First, we allocates an integer array on the GPU that can store all candidate itemsets, i.e., an array of $k \cdot \binom{|\mathcal{L}_{k-1}|}{2}$ elements, where $\mathcal{L}_{k-1}$ is a set of $(k-1)$-PFIs. By allocating the array that accommodates all the candidate k-itemsets, we can generate the candidate itemsets in parallel, without concurrency control. In the merging phase, the GPU checks in parallel whether pairs in $\mathcal{L}_{k-1}$ are joinable. If X_i and X_j are joinable, where X_i is the ith itemset in $\mathcal{L}_{k-1}$, they are combined into a candidate k-itemset. The candidate is stored into the allocated array at a unique index corresponding to i and j. This index is computed as $(|\mathcal{L}_{k-1}| - 1) \cdot i - i \cdot (i + 1)/2 + j - 1$. If X_i and X_j are not joinable, a special value (e.g., -1) is assigned at the same index to indicate that the candidate does not exist. In the pruning phase, if it is confirmed that any $(k - 1)$-subset of a candidate is not contained in $\mathcal{L}_{k-1}$, the special value is assigned at the corresponding index of the candidate as before.

After generating candidates, candidates and non-candidates are intermixed in the resulting array. Thus we filter out the non-candidates by applying the filter primitive. The resulting array is an input to filter and each candidate is considered as an element. The predicate is a function that takes a candidate and returns true if the first element of the candidate is not the special value.

3.3.3 Extracting probabilistic frequent itemsets

Algorithm 2 shows an overview to extract PFIs, which comprises four steps: inclusion check, pruning, filtering, and SPMF computation. *Inclusion check* is to check whether transactions of database include a candidate. By using this result, the pruning step attempts to prune candidates by computing counts and expected supports. Non-pruned candidates become subject to the filtering step. For each candidate, this step filters out transactions that do not include the candidate, because such transactions do not contribute to the SPMF. The final step computes the SPMF of this candidate and determines whether the candidate is a PFI or not. Note that inclusion check and pruning are processed in parallel for all candidates and the rest steps of each candidate are performed in parallel. The following describe the four steps one by one.

Inclusion check

Inclusion check is to check whether transactions of an input database $\mathcal{U}$ include a candidate X. A straightforward implementation of inclusion check using iterative loops with conditional branching deteriorates the performance, because GPUs cannot execute such codes in parallel due to the architecture's limitation. Thus it is more desirable to investigate a method that can be executed efficiently on GPUs.

The information whether a transaction includes a candidate can be represented by one bit. It is therefore a waste of memory to naïvely store the information as an integer element. Instead, we employ a more compact data structure [103]; the result of inclusion check is stored to an array of 32-bit integers, where each integer is treated as a bitstring. The array, which we call *inclist* hereafter, allows for storing the result with $\lceil |\mathcal{U}| \, / \, 32 \rceil$ elements. In addition, the bitstrings enable us to exploit

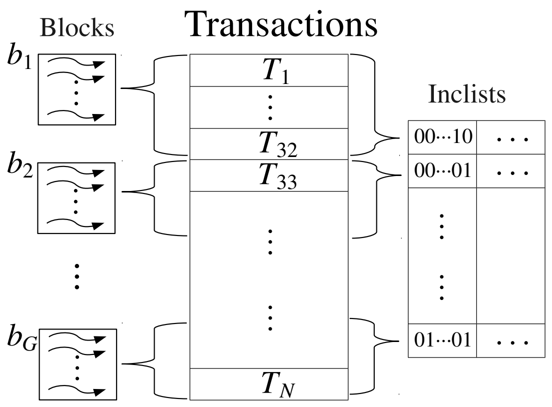


Figure 3.3: An overview to inclusion check for $k = 1$ on the GPU. G is the number of blocks, N is the number of transactions, and there are $|I|$ inclists, where I is a set of all items.

intrinsic functions of GPUs, which are described later.

The computation of inclusion check is divided into two cases according to k, namely, $k = 1$ and $k > 1$. Besides, inclusion check for $k = 1$ can be performed by either the CPU or the GPU. The following explains them one by one.

$k = 1$ (GPU). Figure 3.3 illustrates an overview to inclusion check of GPU approach. Each block of the GPU deals with the same 32 transactions, and each thread in a block is in charge of $|I| / B$ different candidates, where B is the number of threads in one block. In other words, one thread conducts inclusion check of $|I| / B$ candidates with regard to 32 transactions, and packs these 32 boolean values into an element of inclist. We assign threads to 32 transactions so that the threads can store the results without a lock mechanism, because we assume that each element of inclist is a 32-bit integer.

Transactions accessed from a block are transferred to shared memory before inclusion check, because the transactions are shared by the threads within the block and are read multiple times. When transferring the transactions to shared memory, the threads of a warp read consecutive addresses on global memory thanks to the column-major order, as mentioned in Section 3.3.1. These accesses are thereby coalesced into fewer memory reads, so that the transfers can be completed quickly.

For each candidate, a thread judges whether transactions include the candidate by linear search, transaction by transaction. Linear search enables threads of a warp to read the same data on shared memory at the same time. Such memory accesses are broadcasted to all the threads and take only one access time, thus resulting in quite fast memory reads. Having finished the linear search, a thread sets a bit corresponding to a transaction to one, if the transaction includes the candidate; otherwise the thread sets it to 0.

***k* = 1 (CPU).** Since the GPU architecture is based on SIMT, it may be difficult to efficiently parallelize every step of an algorithm. CPUs can be used for parallel processing of such steps instead of GPUs. Inclusion check fits this situation; inclusion check for only $k = 1$ is processed on the CPU and the cases when $k > 1$ are done on the GPU. Since constructing inclists can be highly costly, pruning at $k = 1$ is also performed on the CPU *before* the inclusion check, contrary to Algorithm 2.

The computation of inclusion check and pruning on the CPU proceeds in a different manner than the GPU approach. While the GPU looks up which transactions include candidates, the CPU searches through transactions to find which candidates are contained in the transactions. The major difference from the GPU approach is that transactions are examined only once.

First, the counts and the expected supports of candidate 1-itemsets are computed by scanning the database. With the two values, we judge if candidates can be pruned. Then, inclusion check of non-pruned candidates is carried out as follows. For each item of transaction, we see if the item is still a candidate or not, and if so, we set the corresponding bit of inclist elements. When parallelizing, 32 transactions are handled by one thread to avoid locks as in the GPU approach. Having computed the inclists of non-pruned candidates, the CPU transfers them to the GPU and shifts to parallel processing on the GPU.

***k* > 1.** In this case, it is not necessary to scan the whole database by reusing inclists for $k - 1$, with an idea described in the prior work [34]. The inclist of a candidate k-itemset X is computed as bitwise AND operations between two inclists of $(k - 1)$-PFIs that are used to create the candidate X. For example, the inclist of $\{\mathrm{a}, \mathrm{b}\}$ can be computed as bitwise AND operations between two inclists of $\{\mathrm{a}\}$ and $\{\mathrm{b}\}$. This computation is efficiently parallelizable by assigning a thread to a bitwise AND operation. In addition, this optimization technique substantially reduces accesses to the global memory, thereby making inclusion check run much faster.

Pruning

The second step is to prune out useless candidates. As explained in Section 9, pruning requires counts and expected supports. Having computed counts and expected supports, we can easily test whether candidates are pruned or not. If a candidate is decided to be pruned, a special value is assigned to the first element of the candidate to indicate that it is pruned. After evaluating all candidates, pruned candidates are discarded by applying the filter primitive as in Section 3.3.2. The following describes how to efficiently compute counts and expected supports.

cnt(X). This value means the number of transactions that include the itemset X. The count can be computed by summing the number of bits set to 1 in the inclist of X. The map and reduce primitives are utilized to realize this computation. For the map function, we use a GPU intrinsic function `__popc` [98]. The `__popc` function takes an integer and returns the number of bits that are set to 1 in the binary representation. By applying map with `__popc` to the inclist of X, we can get an intermediate array of partial counts. To this array, the reduce primitive with addition is applied, and cnt(X) is obtained as a result.

esup(X). This value means the expectation of sup(X), which can be computed by summing the existential probabilities of transactions that include the itemset X. At first glance, it may seem that applying the reduce primitive is enough to compute esup(X). However, since the inclusion information is encoded within the bitstrings of inclists, it is difficult to directly apply the primitives. Therefore some specialized function is needed to work with inclists.

To this end, we exploit `__popc` and another GPU intrinsic function `__ffs`. This function takes an integer and returns the position of the first (least significant) set bit [98]. Combining these functions, we design an algorithm to decode a bitstring (Algorithm 3). This algorithm enables us to work only with bits set to one. Since `__ffs` returns the position of the first set bit, we can obtain at the first iteration the index of the first transaction that includes the candidate. Then some operation is done with `acc` as an index, and `x` is shifted right by `pos` bits to make the first set bit disappear. The algorithm repeats these operations `__popc(x)` times (i.e., for all bits set to one).

In order to compute esup(X), we apply the map primitive to the inclist of X with Algorithm 3 as the map function. More precisely, the map function adds up the existential probabilities corresponding to the transactions encoded in the inclist at Line 6 of Algorithm 3, and returns this sum. The reduce primitive aggregates these sums to obtain esup(X).

Filtering

To compute the SPMF of an itemset X, it is sufficient to consider the transactions that include X; the other transactions can be ignored because these transactions do not contribute to the SPMF. In this case, the time complexity of SPMF computation for X decreases from $O(|\mathcal{U}| \log^2 |\mathcal{U}|)$ to $O(\mathrm{cnt}(X) \log^2 \mathrm{cnt}(X))$ [124]. To discard the unnecessary transactions in parallel, we use a procedure similar to the filter primitive.

Algorithm 3: Decode a bitstring.

```
  Input: an integer x
1 n ← __popc(x)                          ▷ the number of bits set to one
2 acc ← −1                               ▷ the position of a current bit in x
3 for $i = 0$ to n − 1 do
4     pos ← __ffs(x)                     ▷ the position of the next bit
5     acc ← acc + pos
6     do some operation using acc as an index
7     x ← x >> pos
8 end
```

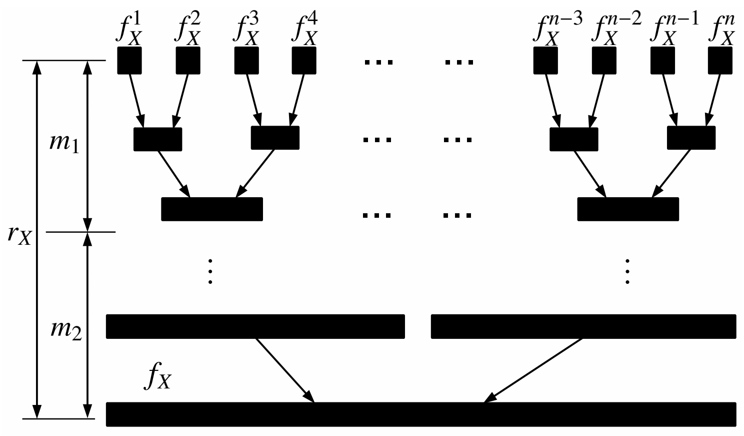


Figure 3.4: A tree structure in computing an SPMF f_X. A node means an SPMF and its size indicates the size of an SPMF.

We firstly decode the inclist of X by using Algorithm 3 and store the result to an array of $|\mathcal{U}|$ elements that contains 0-or-1 values. The elements of this array are flags, indicating whether transactions include the itemset X. This array becomes an input to the scan operation with the addition operator, and the resulting array is the indexes for output. The flag and index arrays are used in the next step to initialize SPMFs.

SPMF computation

Since the computation of SPMFs is the most computationally intensive part of the pApriori algorithm, we have to carefully parallelize this step. When the divide-and-conquer algorithm (Section 9) is adopted, the processing flow of SPMF computation

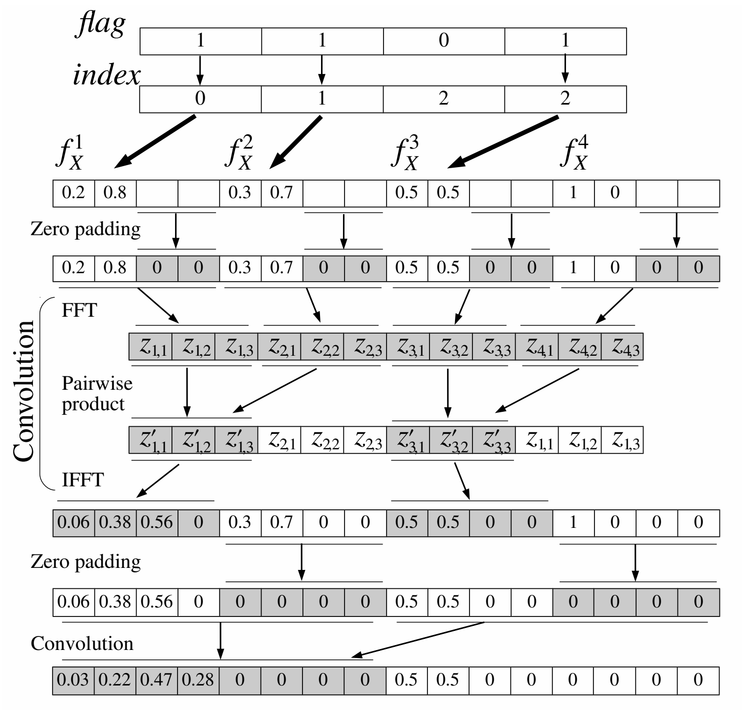


Figure 3.5: The computation of an SPMF $f_{\{b\}}$.

can be illustrated as the tree structure in Figure 3.4. The computation begins from the leaf nodes; by performing convolutions repeatedly, finally the desired SPMF is evaluated. We exploit this structure to achieve high performance on the GPU. To the best of our knowledge, there is no work on this kind of computation.

Before computing the SPMF of an itemset X, we allocate an array F_X of $4 \cdot 2^{\lceil \log_2(\mathrm{cnt}(X)) \rceil}$ elements to store initial SPMFs, which correspond to the leaf nodes f_X^i. In other words, the array F_X stores $2^{\lceil \log_2(\mathrm{cnt}(X)) \rceil}$ (the next-higher power of two to the count) SPMFs of size 4. Hereafter, we call the exponent $\lceil \log_2(\mathrm{cnt}(X)) \rceil$ the *rank* of an itemset X, denoted as r_X. The rank is an important factor because it determines the height of the tree (Figure 3.4), or the time complexity. The time complexity of SPMF computation is expressed with the rank as $O(2^{r_X} \log^2 2^{r_X}) = O(2^{r_X} r_X^2)$.

The reasons that F_X is of size $4 \cdot 2^{r_X}$ are as follows:

1. The number of SPMFs in F_X must be a power of two in order to combine the SPMFs into one SPMF with repeated convolutions.

2. Each SPMF must be able to accommodate a convoluted SPMF, which has three elements after the initial convolution.

3. Making the size of SPMFs a power of two is beneficial to the performance of FFT [99].

The first point makes the number of SPMFs $2^{r_X} = 2^{\lceil \log_2 (\mathrm{cnt}(X)) \rceil}$. The second and third points together lead to the initial SPMFs of size 4.

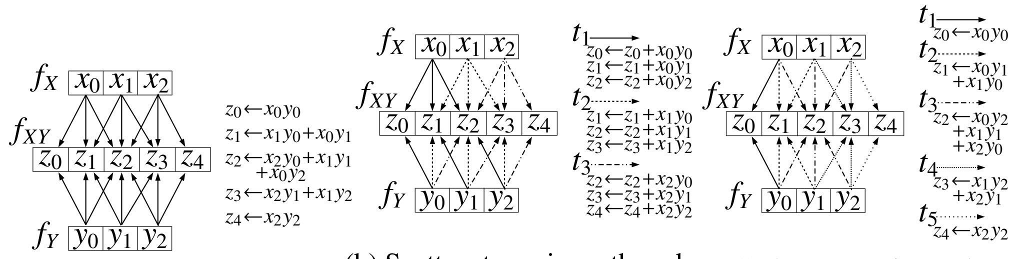
To initialize the array F_X, GPU threads compute the SPMFs of X in databases that consist of only one transaction, by referring to the precomputed *flag* and *index* arrays in the last step. If *flag*[i] is one, the ith thread computes the SPMF with regard to the database that has only the ith transaction. The thread stores the computed SPMF to the array F_X so that it occupies the first two of specific four elements of F_X. The array F_X at this time is the third array from the top in Figure 3.5, which illustrates the computation of an SPMF $f_{\{\mathrm{b}\}}$ with the database in Table 3.1. f_X^i denotes the SPMF of {b} in the database that consists of the ith transaction among those that includes {b}. The first element of f_X^i is the value of $f_{\{\mathrm{b}\}}^i(0)$ and the second element is the value of $f_{\{\mathrm{b}\}}^i(1)$. Note that, in the example, f_X^4 has no corresponding transaction because cnt({b}) is 3, but f_X^4 is added so that there exist a power-of-two number of SPMFs.

To make the size of SPMFs four and correctly compute convolution, zero padding (i.e., padding extra elements with 0) is carried out. Then the convolutions of two adjacent SPMFs on F_X are computed by performing three operations: Fast Fourier Transform (FFT), the pairwise products of two transformed SPMFs, and Inverse FFT (IFFT) on the multiplied ones. The third array from the bottom in Figure 3.5 shows F_X at this point. The shaded areas indicate the elements that are updated with a previous operation. Note that IFFT is only performed on a half of the SPMFs and zero padding is carried out to the other half, thereby enabling next convolutions to take SPMFs of a power-of-two size. The two operations (i.e., zero padding and convolutions) continue until the number of SPMFs becomes one. Finally, the desired SPMF is stored in the array $F_X[0]$ to $F_X[\mathrm{cnt}(X)]$.

As for FFT and IFFT, we used CUFFT, which is the CUDA FFT library [99]. The CUFFT library provides a simple interface for computing parallel FFTs on an NVIDIA GPU. The library achieves the best performance when transforming the 2^a size. By storing the SPMFs in one array with the same size, we can also use batch execution that is a feature of CUFFT.

SPMF computation on shared memory

As mentioned in the previous section, the SPMF computation begins from the leaf nodes in Figure 3.4. By performing convolutions, the size of SPMFs doubles while


f_X
f_{XY}
f_Y
x_0 x_1 x_2
z_0 z_1 z_2 z_3 z_4
y_0 y_1 y_2
$z_0 \leftarrow x_0 y_0$
$z_1 \leftarrow x_1 y_0 + x_0 y_1$
$z_2 \leftarrow x_2 y_0 + x_1 y_1 + x_0 y_2$
$z_3 \leftarrow x_2 y_1 + x_1 y_2$
$z_4 \leftarrow x_2 y_2$
t_1 $z_0 \leftarrow z_0 + x_0 y_0$ $z_1 \leftarrow z_1 + x_0 y_1$ $z_2 \leftarrow z_2 + x_0 y_2$
t_2 $z_1 \leftarrow z_1 + x_1 y_0$ $z_2 \leftarrow z_2 + x_1 y_1$ $z_3 \leftarrow z_3 + x_1 y_2$
t_3 $z_2 \leftarrow z_2 + x_2 y_0$ $z_3 \leftarrow z_3 + x_2 y_1$ $z_4 \leftarrow z_4 + x_2 y_2$
t_1 $z_0 \leftarrow x_0 y_0$
t_2 $z_1 \leftarrow x_0 y_1 + x_1 y_0$
t_3 $z_2 \leftarrow x_0 y_2 + x_1 y_1 + x_2 y_0$
t_4 $z_3 \leftarrow x_1 y_2 + x_2 y_1$
t_5 $z_4 \leftarrow x_2 y_2$

(a) Definition of convolution. (b) Scatter: to assign a thread to an element in an input array. (c) Gather: to assign a thread to an element in a resulting array.

Figure 3.6: Convolution between two SPMFs f_X and f_Y.

the number of SPMFs gets half. In other words, there are a large number of small SPMFs to be convoluted in the beginning of computation. This means that the convolutions can be computed entirely on shared memory of GPUs in the first few levels. Thus we consider dividing the SPMF computation into two phases: shared-memory phase at first m_1 levels and global-memory phase at the rest m_2 levels, as shown in Figure 3.4. In the first phase, the convolutions are directly computed entirely on shared memory, without using FFT. In the second phase, we use the algorithm explained in the previous section. We describe the details of computation on shared memory in the following.

Figure 3.6(a) shows the computation of convolution between two SPMFs denoted as two arrays f_X and f_Y, and the convoluted array is f_{XY}. The arrows from input elements mean contributions to the pointed elements. Resulting elements z_i are calculated as the expressions shown on the right side of the figure.

According to this illustration, we can consider two parallelization options:

Scatter: to assign a thread to an element in an input array.

Gather: to assign a thread to an element in a resulting array.

Figures 3.6(b) and 3.6(c) depict the two options, respectively. In the figures, computations by different threads t_i are drawn in different line styles. Pairs of a thread and its work are also shown on the right side. For example, the thread t_1 in Figure 3.6(b) is assigned to x_0 and computes $z_0 \leftarrow z_0 + x_0 y_0, z_1 \leftarrow z_1 + x_0 y_1$, and $z_2 \leftarrow z_2 + x_0 y_2$, while the thread t_1 in Figure 3.6(c) is assigned to z_0 and computes $z_0 \leftarrow x_0 y_0$.

Pros and cons of the two options are twofold: memory write conflict and load balance. As can be seen from Figure 3.6(b), the three threads of the scatter approach perform the same amount of work and thus this approach provides load balancing. However, it is possible that different threads update the same address at the same time and a conflict occurs. Such conflicts must be resolved by using atomic functions available on GPUs, at the expense of performance. On the other hand, write conflicts never occur with the gather approach because threads are assigned to elements of a *resulting* array, although load balance is not achieved as shown in Figure 3.6(c).

During the SPMF computation on shared memory, two arrays are used to maintain SPMFs. The initial SPMFs handled by a specific block are stored in one of the arrays, by packing the SPMFs into the array. Two adjacent SPMFs in the array are convoluted and the results are stored in the other array. Then, in turn, the latter array becomes the next input array. The convolutions are computed in this fashion until we reach the specified m_1 level.

Simultaneous computation of multiple SPMFs

The algorithm of SPMF computation described so far cannot fully exploit the parallelism of GPUs when the rank of a candidate is low. To alleviate the problem, this section introduces a technique of simultaneous computation of multiple SPMFs. This technique allows for the effective utilization of GPUs even if there are only candidates with low ranks.

The main idea is simple: to pack the arrays F_X of multiple candidates into a single array. Then we can perform FFT to multiple candidates in parallel by using the feature of batch execution in CUFFT [99]. Zero padding and pairwise products are also carried out simultaneously on multiple SPMFs. The targets of packing in one batch are candidates with the same rank; otherwise the number of convolutions to be performed disagrees among candidates.

The last loop of Algorithm 2 is replaced with Algorithm 4, which shows a pseudo-code of the technique. Candidates are firstly sorted by their counts, to put candidates with the same rank side by side (Line 1). Then we continue to get the next candidates (Lines 5–7) while the rank of next candidate is the same with the current one and the buffer for computation has enough space (Line 8). Having chosen candidates, the algorithm initializes the buffer with the candidates so that the computation of SPMFs can be started. Finally, it computes multiple SPMFs simultaneously and tests whether the current candidates are a PFI or not. The iteration continues until there is no more candidate.

Algorithm 4: Simultaneous computation of multiple SPMFs.

```
   Input: a set $C$ of candidates
1  Sort the candidates $C$ by their counts
2  while $C \neq \emptyset$ do
3      $S \leftarrow \emptyset$                          ▷ the target candidates at this iteration
4      do
5          $X \leftarrow$ the next candidate in $C$
6          $S \leftarrow S \cup \{X\}$
7          $C \leftarrow C - \{X\}$
8      while $C \neq \emptyset \wedge$ the rank of next candidate $= r_X \wedge$ buffer $B$ has enough
         space
9      Initialize the buffer $B$ with the candidates in $S$
10     Compute SPMFs using the buffer $B$
11     Test whether the candidates in $S$ are PFIs or not
12 end
```

Table 3.2: Characteristics and parameters of datasets.

Dataset	# of items	# of transactions	Density	Probability distribution	Default `minsup`
connect	129	67,557	33%	$\mathcal{U}(0,1)$	48
accidents	468	340,183	7.2%	$\mathcal{N}(0.5,0.02)$	35
T25I10D500K	7,558	499,960	0.33%	$\mathcal{U}(0,1)$	0.65
kosarak	41,270	990,002	0.020%	$\mathcal{N}(0.5,0.02)$	0.3

3.4 Experiments

We conducted extensive experiments to comprehensively evaluate our GPU-based implementation. Section 3.4.1 summarizes the setting of experiments and datasets. Sections 3.4.2–3.4.4 show the results on each step of the GPU-based algorithm, namely, inclusion check, pruning, and SPMF computation. Section 3.4.5 shows the result on the whole algorithm.

3.4.1 Experimental setup

We implemented our scheme using CUDA 5.5 and conducted experiments on an NVIDIA Tesla K20X, which has global memory of 6 GB and 14 SMs, each of which consists of 196 SPs at 732 MHz. The CPU is a hexa-core Intel Xeon Processor E5-1650 v2 at 3.50 GHz with memory of 32 GB. It supports Hyper-Threading Technology and can run 12 threads in total.

For a fair comparison, we parallelized the pApriori implementation that is made available online by the authors [124]. OpenMP[2] was used to parallelize pApriori on the CPU. In order to optimize FFT computation, we employed the FFTW[3] library with the option of SIMD support for further acceleration. SPMFs are computed on the CPU by using the same algorithm in Section 8. Although the computation of a single SPMF is not parallelized, the SPMFs of different candidates are computed in parallel by multiple threads. While SPMFs are small, the dynamic-programming algorithm [124] is adopted instead of divided-and-conquer one. This was the fastest way of computing SPMFs in our preliminary experiments. The technique of simultaneous computation of multiple SPMFs was not considered because its objective is to give the GPU enough parallelism to saturate. On average, our implementation is 30 times faster than the original one. In the following sections, we only show the result of our CPU implementation.

We used four datasets for the experiments. Table 3.2 summarizes the characteristics and the parameters of the datasets. The density of a dataset is computed as the average length of transactions divided by the number of items. The three datasets *connect*, *accidents*, and *kosarak* are real datasets that are accessible on Frequent Itemset Mining Implementations (FIMI) Repository[4]. The dataset *T25I10D500K* is a synthetic dataset generated by IBM data generator [124]. While connect is the smallest and densest, kosarak is the largest and sparsest, and the other two datasets lie in the middle. Existential probabilities for connect and T25I10D500K are given by a uniform distribution between 0 and 1; probabilities for accidents and kosarak are randomly drawn from a normal distribution with mean 0.5 and variance 0.02. The default values of `minsup` are used in our experiments unless explicitly specified. In the following, the result when changing `minprob` is not shown and 0.5 is used because it has little impact on execution time. Figure 3.7 shows the number of non-pruned candidates for each rank on the four datasets. We shall refer to this figure when needed.

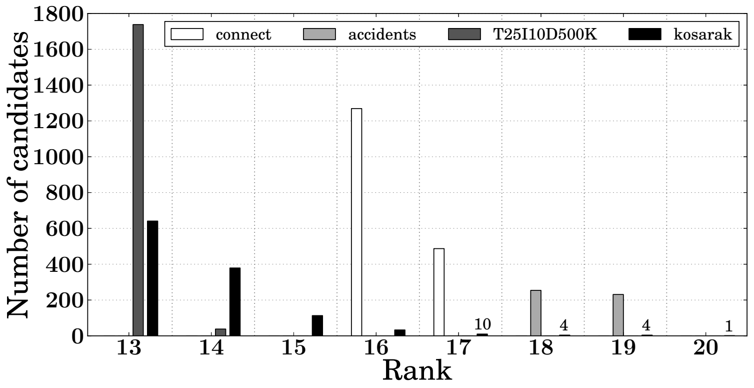
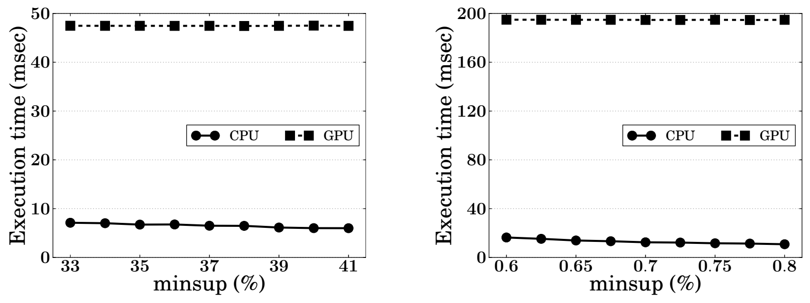


Figure 3.7: The number of non-pruned candidates that have a corresponding rank.



(a) accidents (b) T25I10D500K

Figure 3.8: Execution times of inclusion check for $k = 1$.

3.4.2 Results on inclusion check

In this section, we evaluate the approaches to inclusion check. First, we compare the CPU approach (`CPU`) and the GPU approach (`GPU`) for the case of $k = 1$. Then the two methods for $k > 1$ are evaluated: One is a method without bitwise AND operations (`Baseline`) and the other uses bitwise operations (`Bitwise AND`).

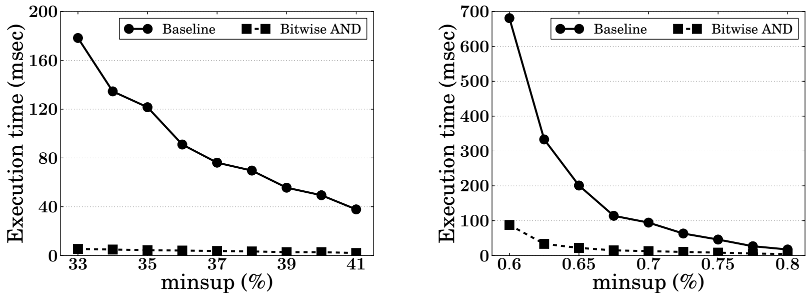
$\boldsymbol{k = 1}$**.** Figure 3.8 shows the execution time with varying `minsup` values, which are the percentages to the number of transactions. The execution time of `CPU` includes the transfer time of inclists. We omit the result on connect because it is similar to the result on accidents. Also the result on kosarak is not shown because `GPU` did not work with it due to running out of memory.

The CPU approach is considerably faster than the GPU counterpart. Specif-

[2]`http://openmp.org/`

[3]`http://www.fftw.org/`

[4]`http://fimi.cs.helsinki.fi/`



(a) accidents (b) T25I10D500K

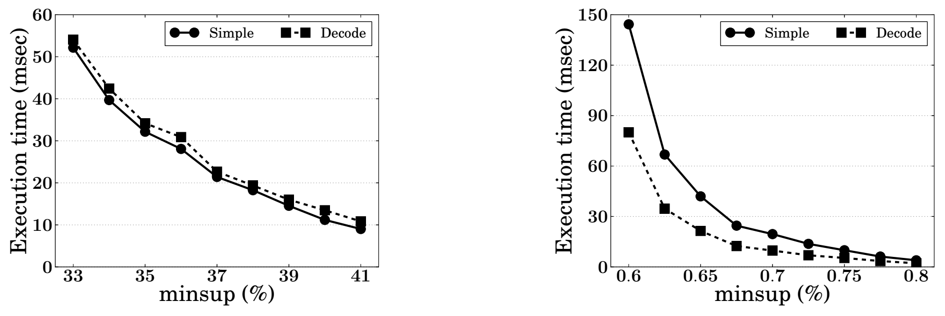
Figure 3.9: Execution times of inclusion check for $k > 1$.

ically, CPU is 7 and 12–19 times faster than GPU on accidents and T25I10D500K, respectively. The result reveals that GPU is particularly inefficient on T25I10D500K. This is because GPU needs to read transactions as many times as the number of items, whereas CPU reads transactions only once.

Although we may adapt the scheme of CPU to a GPU implementation, it is difficult to efficiently implement it due to the limitation of size of shared memory. In addition, we can consider that the CPU approach already realizes sufficient performance, because inclusion check is not a dominant step in the whole algorithm (Section 3.4.5). Thus we did not try the possibility. The lesson learned here is, as is well known, that it is easier and more efficient to let CPUs do the work that is difficult to parallelize on GPUs.

$k > 1$. Figure 3.9 depicts the total execution time of inclusion check for $k > 1$ with varying `minsup` values. `Baseline` uses the same scheme with the GPU approach for $k = 1$, instead of the faster CPU approach because it is specialized for the $k = 1$ case. We omit the results on connect and kosarak for the same reasons as before.

`Bitwise AND` achieves 18–32x speedups on accidents and 4–10x speedups on T25I10D500K. As the `minsup` value decreases, the bitwise method gains more and more performance improvements. The reason is that a small `minsup` value leads to the larger number of candidates and thus `Baseline` reads transactions more times. Meanwhile, the bitwise method is more effective when using accidents, because the dataset has candidates with higher ranks (Figure 3.7). This means that the baseline approach needs to set more bits to construct inclists, resulting in higher processing cost. In particular, `Bitwise AND` outperforms `Baseline` by a factor of 32.9 when `minsup` is 33 % on accidents.



(a) accidents (b) T25I10D500K

Figure 3.10: Impact of Algorithm 3 for pruning on the performance.

3.4.3 Results on pruning

This section evaluates the effect of the decoding algorithm (Algorithm 3) with the comparison to the algorithm that simply loops 32 times to manipulate each bit. Figure 3.10 illustrates the execution time as a function of `minsup` on two datasets. Since the results on connect and kosarak are similar to the results on accidents and T25I10D500K, respectively, we omit them here.

The decoding algorithm speeds up the pruning on T25I10D500K (Figure 3.10(b)), while it is slightly slower on accidents (Figure 3.10(a)). The reason of this behavior can be explained as follows: Efficiency of Algorithm 3 depends on the number of bits set to 1 in an input integer. If few bits are set to one, the algorithm can process only bits of interest with fewer instructions. On the other hand, if most of the bits are set to one, the algorithm performs a few extra instructions compared to simple loops. Specifically, accidents has high-rank candidates, resulting in inclists with the large portion of bits set to one; T25I10D500K has only low-rank candidates, leading to inclists with few bits set to one. Consequently, the algorithm exhibit the performance as shown in Figure 3.10.

3.4.4 Results on SPMF computation

We measured the computing time of SPMFs on the four datasets. Figure 3.11 shows the speedups of GPU implementations with the comparison to the CPU. Various optimization techniques are incrementally added to a naïve GPU implementation, including filtering, computation on shared memory, and simultaneous computation. In the following, we evaluate the effect of each technique one by one.

Filtering. The naïve GPU implementation, denoted as `GPU` in Figure 3.11, uses

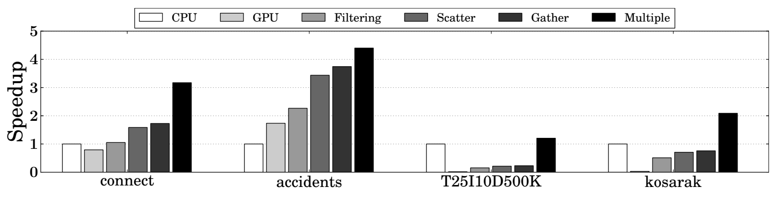


Figure 3.11: Speedup values of SPMF computation compared to CPU with various optimization techniques.

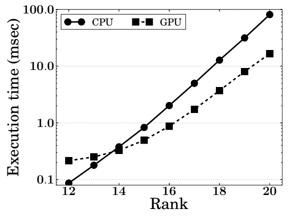


Figure 3.12: Rank vs. computing time of SPMF.

the algorithm in Section 8 without filtering. GPU shows worse performance than CPU except for the accidents dataset. By applying filtering, we can achieve substantial speedups, especially when using T25I10D500K and kosarak. For example, filtering speeds up GPU 16.9 times on kosarak. This is because the two datasets generate a large number of low-rank candidates (Figure 3.7). Recall that filtering reduces the time complexity of SPMF computation for an itemset X from $O(|\mathcal{U}| \log^2 |\mathcal{U}|)$ to $O(\mathrm{cnt}(X) \log^2 \mathrm{cnt}(X))$. Thus we can greatly decrease the computational cost by filtering on T25I10D500K and kosarak. On the other hand, on connect and accidents, the speedup by filtering is only around 1.3, because these datasets produce only high-rank candidates (Figure 3.7). At this point, the GPU is still slower than the CPU except for accidents.

Shared memory. The next applied technique is to compute SPMFs on shared memory. There are two variants of this technique: scatter and gather. Both of them require setting the m_1 value in Figure 3.4; this was empirically determined to $m_1 = 7$ and $m_1 = 8$ for scatter and gather, respectively.

Figure 3.11 reveals that gather is always faster than scatter on the present datasets. While the gather approach computes without write conflicts at the expense of load imbalance, the scatter approach uses atomic functions to resolve such conflicts. In other words, scatter needs to serialize memory accesses if conflicts occur. Since memory accesses are much slower operations than computation in general, gather leads to the faster execution. Compared with `Filtering`, the gather kind of computation on shared memory achieves around 1.5 times speedups. Nevertheless, when T25I10D500K or kosarak is used, the GPU is not yet faster than the CPU.

Multiple SPMFs. The last optimization is the simultaneous computation of multiple SPMFs. This technique is particularly effective on T25I10D500K; it achieves 5.2x performance compared with `Gather` and finally the GPU can outperform the CPU on this dataset. In general, the technique is beneficial when a dataset has many low-rank candidates. Thus `Multiple` is 2.8x faster than `Gather` on kosarak,

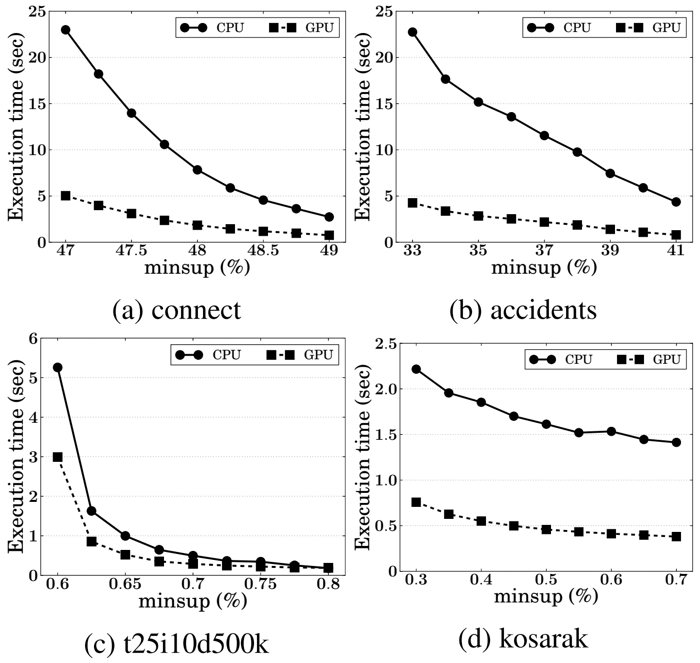


Figure 3.13: Execution times of probabilistic frequent itemset mining with varying `minsup` values.

whereas the speedup on accidents is merely 1.2x since it has only high-rank candidates.

Total speedup. Equipped with all the techniques, the GPU outperforms the CPU by a factor of 3.17, 4.40, 1.20, and 2.09 on connect, accidents, T25I10D500K, and kosarak, respectively. So as to see why there are a large gap of speedup values among datasets, we show the computing time of SPMFs as a function of rank in Figure 3.12. We here measured the average time over computing 2^{10} SPMFs of candidates with a corresponding rank. This figure discloses that the GPU is slower than the CPU until the rank of 13 and then the GPU gets faster and faster than the CPU as the rank increases. In other words, the GPU is more efficient for high ranks. That is why the GPU is 4.40x faster than the CPU on accidents, which has only high-rank candidates. By contrast, the GPU shows the almost same performance with the CPU on T25I10D500K, which has only low-rank candidates. The slow performance of GPU at low rank is due to the filtering time; filtering takes time depending on the number of transactions, regardless of ranks.

Table 3.3: Time breakdown of CPU and GPU execution. The numbers are in milliseconds. The “Inclusion Check” part of CPU incurs inclusion check, pruning, and filtering because they are simultaneously run on the CPU. The “Others” part of CPU means candidate generation. The “Others” part of GPU includes CUDA initialization, candidate generation, and data transfer between the CPU and GPU.

		Inclusion Check	Pruning	Filtering	SPMF	Others	Total
accidents							
	CPU	2933			11547	588.1	15068
	GPU	2.993	33.71	49.03	2582	173.5	2841
T25I10D500K							
	CPU	471.3			364.0	125.4	960.7
	GPU	19.87	19.96	185.9	107.2	188.7	521.6

3.4.5 Results on probabilistic frequent itemset mining

Finally, we compare the whole processing of probabilistic frequent itemset mining. Figure 3.13 shows execution time vs. `minsup` on the four datasets. Time breakdowns of CPU and GPU executions on accidents and T25I10D500K are also summarized in Table 3.3.

Figures 3.13(a) and 3.13(b) show the results on connect and accidents, respectively. The GPU outperforms the CPU by a factor between 3.6 and 5.5. This performance results from the speedup of SPMF computation as well as the other steps. For example, when `minsup` is 35 % on accidents, the CPU execution spends about 3 seconds for inclusion check, whereas the GPU performs the same computation in 85 milliseconds (Table 3.3). In addition, the GPU can compute SPMFs more than 4 times faster than the CPU. These improvements together lead to the high performance of our GPU implementation.

Figure 3.13(c) illustrates the result on T25I10D500K. In this case, the GPU is 1.3–1.9 times faster than the CPU when `minsup` is 0.6–0.75 %, while the GPU is slightly slower than the CPU with `minsup` of 0.8 %. This performance degradation is due to the overhead of CUDA initialization, which takes approximately 100 milliseconds (i.e., more than half of the total execution time). As the `minsup` values decreases, actual processing times get longer and performance improvement with the GPU becomes apparent. However, the result on T25I10D500K does not show speedups to the same extent as the results on connect and accidents. This is because most of the candidates have ranks less than 14. As explained in Section 3.4.4, SPMF computation on the GPU for the rank below 14 is slower than that on the CPU. Consequently the GPU is not so efficient on this dataset. Nevertheless, our GPU implementation outperforms the CPU because of the speedup of other steps as shown in Table 3.3.

The result on kosarak is shown in Figure 3.13(d). The GPU achieves the speedup ratio of 3.7–5.3, although this dataset generates many low-rank candidates. The reason is that the dataset also generates candidates with very high ranks (Figure 3.7). In addition, the steps other than SPMF computation are much improved because the parallelism of the GPU is fully utilized due to the large size of dataset.

3.5 Summary

This chapter has presented a method of probabilistic frequent itemset mining using the GPU based on the pApriori algorithm. The main idea is to accelerate SPMF computation by employing the filter primitive and exploiting their tree-structured computation with shared memory on the GPU. The SPMF computation is further accelerated by handling multiple SPMFs simultaneously. We have also presented an algorithm to efficiently manipulate a bitstring and utilized several data-parallel primitives to achieve high parallelism in other parts of the algorithm. We implemented our methods using CUDA and compared the performance with parallelized pApriori. With the comparison to parallelized pApriori, our implementation outperforms by a factor of up to 5.5.

Chapter 4

GPU-Accelerated Comparison Sorting

Sorting is a fundamental operation in computer science, especially database systems, and its acceleration has significant importance [39, 68]. In particular, *comparison sorting* is paid much attention because of its wide applicability. Comparison-sorting algorithms sort an input sequence by only comparing pairs of elements [68]. The algorithms can be applied to any kind of keys if the keys are comparable. Thus there have been a large number of comparison-sorting algorithms using GPUs. One of the fastest comparison-sorting algorithms on GPUs is an algorithm based on merge sort. This algorithm, however, has the limitation that it slows down when sorting large-scale data. This chapter presents a novel GPU-based algorithm suited for sorting large-scale data. The algorithm achieves high performance by first partitioning the data into multiple buckets and sorting the buckets cooperatively.

4.1 Introduction

Sorting is a fundamental operation in computer science, and it is widely utilized in many algorithms and applications such as database systems [39, 68]. Thus, acceleration of sorting is significantly important for speeding up a wide range of applications. One of the accelerating ways is to employ parallel processing capability of modern processors. In particular, acceleration by GPUs has recently attracted much attention, and many approaches have been proposed [8, 73, 92, 109].

Currently, two state-of-the-art methods for GPU sorting exist: (1) radix sort by Merrill and Grimshaw [92] and (2) merge sort by Baxter [8], hereafter referred

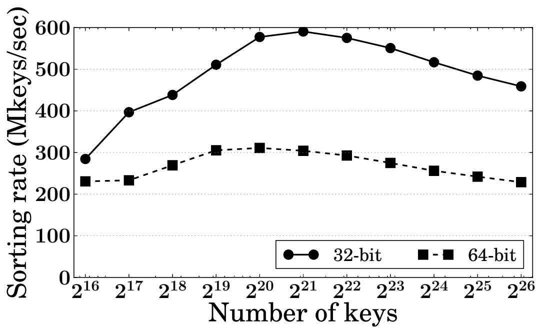


Figure 4.1: The throughput of Baxter sort [8]. The x-axis shows the number of input keys, and the y-axis means the throughput (i.e., the number of sorted keys per second).

to as Baxter sort. Radix sort is faster when sorting 32-bit keys, but it is slow for larger keys such as 64-bit keys, because its computational complexity depends on the key length. Moreover, since radix sort relies on a bit representation of keys, it is applicable to only limited kinds of keys. One the other hand, merge sort is a comparison-sorting algorithm, which can be applied to any kind of keys if the keys are comparable. Thus, we focus on merge sort in this study for the sake of wider applicability.

Baxter sort is the fastest comparison-sorting algorithm. However, it has the limitation that the performance degrades when the number of keys to be sorted exceeds some point. Figure 4.1 shows the performance of Baxter sort in terms of throughput (i.e., the number of sorted keys per second). The data is arrays generated from a uniform distribution. The figure shows that the performance is highest when the number of keys is 2^{20}, and it becomes lower for the larger number of keys. This behavior is due to the increasing number of merge passes. To sort the data with n keys, merge sort requires $\log n$ merge passes. This means that the number of merge passes increases as the data becomes large. In other words, the processing time becomes more than two times longer when the data size doubles.

This chapter presents a novel algorithm for efficiently sorting larger data on GPUs. The algorithm is based on two existing algorithms, namely samplesort [73] and Baxter sort [8]. The proposed approach reduces the number of merge passes by firstly partitioning the input data into k buckets by the partitioning method of samplesort. Then all of the buckets are sorted by merge sort in parallel. Partitioning can reduce the number of merge passes by $\log_2 k$, thereby improving the overall performance. For sorting buckets, it is important to take into account the variability of bucket sizes. If we simply assign one GPU block to one bucket, it may suffer

Algorithm 5: Samplesort

```
Input: An array $A$ to be sorted
if $|A| > M$ then
    $S \leftarrow$ extract a random sample of $\alpha k$ elements from the array $A$
    Sort $S$
    $(s_0, s_1, ..., s_k) \leftarrow (-\infty, S_{\alpha}, S_{2\alpha}, ..., S_{(k-1)\alpha}, \infty)$   ▷ $s_i$ is the $i\alpha$th elements in $S$. $s_0$ and $s_k$ are sentinels representing minimum and maximum values, respectively.
    for $x \in A$ do
        Find $i$ such that $s_{i-1} \leq x < s_i$
        Insert $x$ into the bucket $B_i$
    end
    Apply samplesort to each bucket $B_i$
    $A \leftarrow$ combine the buckets $B_1, ..., B_k$
else
    Sort $A$ by another algorithm
end
return $A$
```

from the load-imbalance problem. To address this issue, our method adaptively assigns multiple blocks to one bucket according to the bucket sizes. Experimental evaluation shows that our proposal outperforms Baxter's merge sort by 39%.

4.2 Preliminaries

This section describes samplesort and Baxter sort, which base our algorithm.

4.2.1 Samplesort

Samplesort [13] is a partitioning-based sorting algorithm. It recursively partitions the input data into k buckets, and if the size of a bucket becomes sufficiently small, the bucket is sorted by another algorithm (e.g., insertion sort). Algorithm 5 shows a pseudo-code of samplesort. If the size of the input array A is smaller than a threshold M, the array is sorted by some sorting algorithm (Line 12). Otherwise, it partitions the data A into k buckets and recursively applies the algorithm to the buckets (Lines 2–10). For partitioning, $k - 1$ splitters are selected from the input array A. To this end, αk elements are sampled from A (Line 2), where α is an oversampling factor and a larger α leads to buckets with more balanced sizes. Subsequently the sample is sorted, and splitters are selected from the sample (Lines 3–4). On the basis of these splitters, the elements in the array A is assigned to buckets. For an element x, if $s_{i-1} \leq x < s_i$, x is assigned to the bucket B_i (Lines 6–7). Having finished partitioning, samplesort is applied to each bucket (Line 9).

Leischner et al. [73] proposed a GPU implementation of samplesort. Their implementation partitions the input array into buckets in parallel until all of the buckets have smaller sizes than a threshold, by using several data-parallel primitives. Then one bucket is sorted by one thread block. A limitation of this implementation is load imbalance due to the difference of bucket sizes. Moreover, if a bucket is too large or too small, the compute resources of GPUs are not well utilized.

4.2.2 Baxter sort

Baxter [8] presented a GPU implementation of merge sort, which we call *Baxter sort*. This implementation exploits a high-performance merge method based on *merge path* [44, 100]. Merge path partitions merge of a pair of sorted lists into p equi-sized subproblems, thereby enabling completely load-balanced merge.

Baxter sort consists of the following two phases:

1. Intra-tile sort: blocks sort distinct parts (or *tile*) of the input array.
2. Cooperative merge: merge pairs of the sorted tiles until only one tile remains.

Figure 4.2 illustrates a schematic of Baxter sort. The input array is divided into multiple tiles, and each tile is sorted by one block. The size of tiles should be set to an appropriate value according to the GPU and characteristics of data. Having finished intra-tile sort, the tiles are merge by multiple blocks cooperatively. Merge of one level is performed in two steps: (1) merge path partitions the input so that the load-balanced merge is done in parallel; and (2) each block merge a distinct portion of a pair of tiles. As indicated in Figure 4.2, the number of blocks is constant throughout sorting, and each block processes the same amount of elements. Also note that the number of blocks per merge increases as the tile size becomes larger.

4.3 Proposed method

The proposed method consists of two phases: (1) data partitioning: the input data is partitioned into k buckets; and (2) intra-bucket sort: each bucket is sorted in parallel by merge sort. The overview of our method is shown in Figure 4.3. The first phase

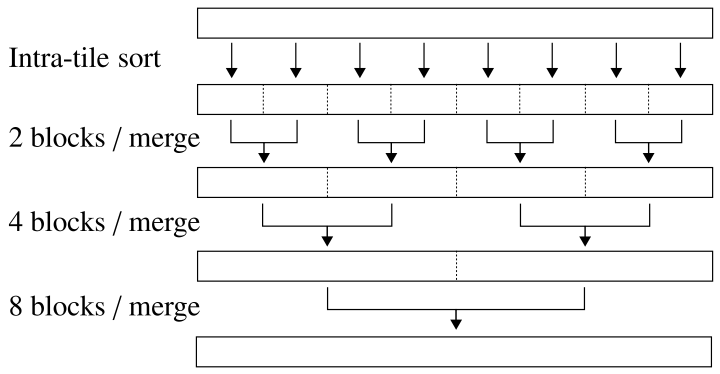


Figure 4.2: Overview of Baxter sort.

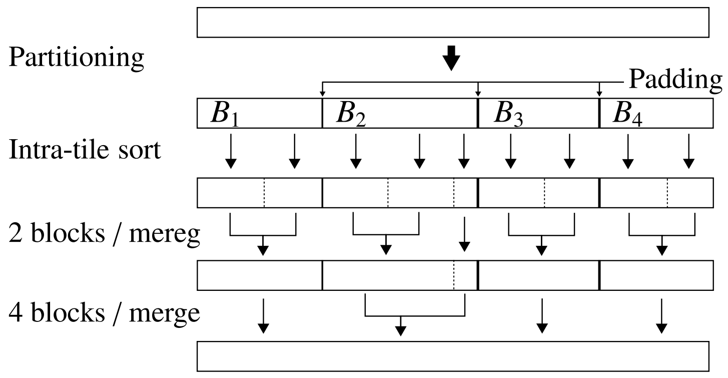


Figure 4.3: Overview of our method.

partitions the data by using the algorithm of samplesort, and the second phase sorts each bucket by Baxter sort. On the one hand, samplesort has the advantage that partitioning can drastically reduce the problem size, but the later phase of samplesort tends to suffer from the load imbalance issue. On the other hand, merge sort is able to achieve load balanced sorting, but it has the disadvantage that the required number of merge passes can become large. Our algorithm combines the two algorithms, fusing both advantages.

4.3.1 Data partitioning

This section describes a data partitioning method. This method is basically same to the method by Leischner et al. [73], but several improvements are introduced. The algorithm consists of four steps: (1) selecting splitters, (2) histogram, (3) scan, and (4) scatter.

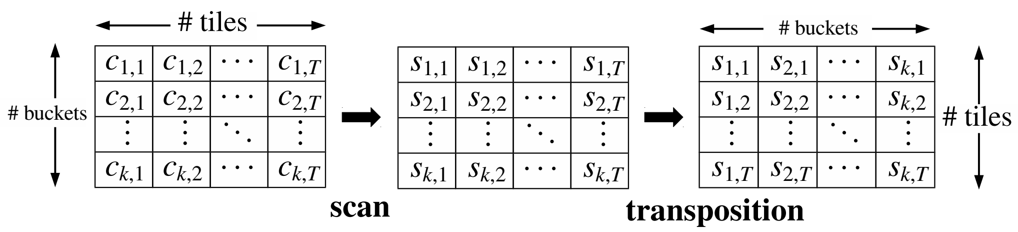


Figure 4.4: Scan and matrix transposition. The entry $c_{i,j}$ is the number of keys contained in the bucket i within the jth tile.

Selecting splitters. The first step select $k - 1$ splitters from the input. As described in Section 4.2.1, this step is comprised of three sub-steps: (1) to extract a sample of αk elements, (2) to sort the sample, and (3) to select splitters. These sub-steps are implemented by one kernel, using only one block. The block obtains sample elements from the input, store the sample to shared memory, and sort them. Then the $i\alpha$th element in the sorted sample becomes the ith splitter. As for the oversampling factor α, Leischner et al. use some constant value, while we use the maximum possible value for better balancing of bucket sizes. For example, if shared memory is 48 KB, $k = 128$, and the input is 32-bit integers, then $\alpha = 48 \cdot 1024 \;/\; (4 \cdot k) = 96$. The extracted splitters are maintained by binary search tree, as proposed by Leischner et al. [73]. This format enables us to efficiently search the corresponding bucket of an element.

Histogram. The second step computes the histogram that maintains the sizes of k buckets, on the basis of splitters selected in the previous step. By applying the scan primitive over this result, we can determine the position of each bucket after partitioning.

One block process a distinct part (or tile) of the input, and computes the histogram of this tile. The histogram is maintained on shared memory for faster accesses from the block. Each thread examines one element from the tile, and decide which bucket the element belongs to by using the binary search tree, constructed in the previous step. Then the corresponding entry of the histogram is incremented by 1. Since this increment may be simultaneously performed by multiple threads, we use an atomic function for concurrency control.

The computed histograms can be regarded as a matrix shown at the left in Figure 4.4, where an entry $c_{i,j}$ means the number of keys from the jth tile that belong to the ith bucket. For the later steps, the histograms are stored on global memory such that the sizes of the same bucket are stored consecutively. In other words, the matrix is stored in row-major order.

Scan. The third step applies prefix sum over the histograms created in the second step. In this step, we introduce two optimization techniques for better global-memory accesses: (1) *matrix transposition* and (2) *bucket alignment*.

The first technique is matrix transposition, transposing the histogram matrix in order to maximize the opportunity of coalesced accesses. In the next step, a block that processes the jth tile need to read the elements $s_{i,j}$ ($1 \leq i \leq k$) of the result of scan. In other words, the block requires the jth column of the matrix at the middle in Figure 4.4. However, since this matrix is stored in row-major order, warps access non-consecutive memory addresses, and thus coalesced accesses are not achieved. On the other hand, by transposing the matrix from row-major order to column-major order, we can fully achieve coalesced accesses.

The second technique is bucket alignment, which is also related to coalesced accesses. Coalesced accesses occur when all threads of a warp access an aligned and contiguous region of 128 bytes. However, the first element of each bucket is placed on not the boundary of a 128-byte segment but an arbitrary location. This implies that, when a warp accesses the first elements of a bucket, the accesses are performed by two or more transactions. To address this issue, bucket alignment adjusts the address of the first element of each bucket so that each bucket starts from the boundary of a 128-byte segment. This alignment can be easily realized by modifying the elements $s_{i,1}$ ($1 \leq i \leq k$) of scan result.

Scatter. The fourth step is scatter, relocating the data elements to corresponding buckets. The input array is divided into tiles in the same way as the second step. A block processes a tile, computing again the histograms. This is because re-computation is faster than storing and reading the result computed in the second step [73]. When checking which bucket an element belongs to, a thread increments the corresponding histogram element by an atomic function, which returns the original value. With this value and the scan result, we can determine the index that the element should be outputted.

If we simply output the elements to global memory, warps do not satisfy the condition of coalesced accesses, resulting in slow accesses. Thus, before writing to global memory, we arrange the elements on shared memory so that the elements of the same bucket occupy a contiguous area of shared memory. Although Leischner et al. [73] describe that it is faster to directly output the elements to global memory, we adopt arranging elements on shared memory because our preliminary evaluation showed that this scheme is faster than direct outputting.

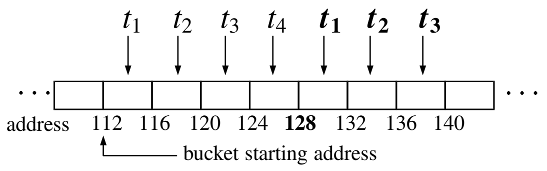

t_1 t_2 t_3 t_4 $\boldsymbol{t_1}$ $\boldsymbol{t_2}$ $\boldsymbol{t_3}$
address 112 116 120 124 **128** 132 136 140
bucket starting address

Figure 4.5: An optimization for de-alignment. This figure illustrates the situation that multiple threads t_i accesses elements of an array, where t_i is the ith thread of a block.

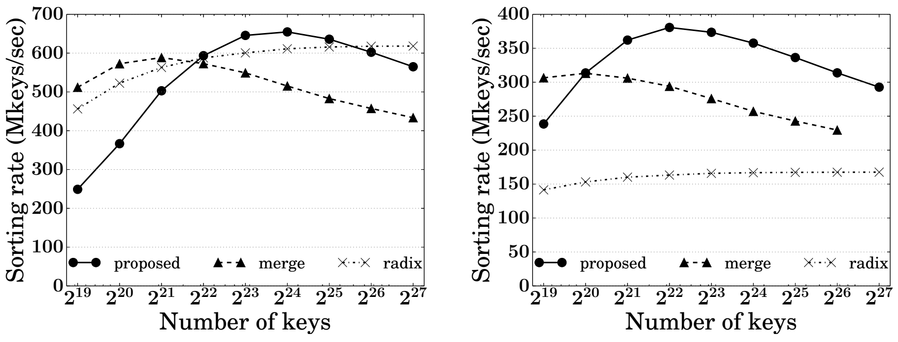
4.3.2 Intra-bucket sort

This section describes the second phase of our proposed method, intra-bucket sort. This phase mainly consists of three steps: (1) block assignment, (2) cooperative merge, and (3) bucket de-alignment. Block assignment allocates thread block IDs to buckets. This step is needed because buckets have different sizes, and thus they are processed by different numbers of blocks, given that blocks handle the same number of elements. On the basis of this assignment, the merge step is performed, sorting the elements within buckets. Finally, since bucket addresses have been modified by bucket alignment, we need to “de-align” the buckets.

Block assignment can be realized by using the result of scan in the data-partitioning phase. From the scan result, the array of bucket sizes are created. On the basis of this array, we compute how many tiles the buckets are divided into, and also compute the scan of this result. By computing these arrays, we can obtain the following information of each block: (1) which bucket should be processed; and (2) how many blocks processing the same bucket exist before this block.

The second step sorts the elements within the buckets. While this step is basically same as Baxter sort, the difference is that we need to take into account the boundaries of buckets. As with Baxter sort, intra-tile sort is firstly performed, and then adjacent tiles within the same buckets are merged. Note that, if the corresponding tile does not exist (e.g., see the bucket B_2), merge is not performed and the tile is just outputted without modification. The merge pass is repeated until all the buckets are sorted.

At the last merge pass, we need to “de-align” the buckets, i.e., return the modified bucket positions to the original ones. This can be easily achieved by outputting the merge results to the original addresses. However, a simple implementation suffers from non-coalesced accesses because write operations of warps cross the boundary of 128-byte segments. To resolve this issue, we separately handle write operations to addresses before and after a boundary. Specifically, writes to ad-



(a) 32-bit integers (b) 64-bit integers

Figure 4.6: Throughput on data generated by a uniform distribution. The x-axis shows the number of keys and the y-axis shows the throughput.

dresses before a boundary are performed by a small number of threads, and writes to addresses after a boundary are carried out normally (Figure 4.5). This scheme enables us to satisfy the condition of coalesced accesses for the write operations after boundaries.

4.4 Experiments

We conducted experiments to evaluate the performance of our proposed method on a server equipped with NVIDIA Tesla C2050, which has global memory of 3 GB and 14 SMs, each of which includes 32 SPs at 1.15 GHz. We compare the following three implementations: (1) our proposed method (*proposed*); (2) Baxter sort (*merge*) [8]; and (3) Merrill's radix sort (*radix*) [94]. Unless otherwise noted, the input datasets are randomly generated with uniform distribution.

4.4.1 Performance comparison

Figure 4.6 shows the sorting throughput of the three implementations with various numbers of keys and different key sizes. The proposed method outperforms Baxter sort when the data is large enough. Specifically, in the case of 32-bit keys, *proposed* becomes faster when the number of keys is larger than 2^{22}. In particular, when the number of keys is 2^{25}, *proposed* achieved 32% better throughput than *merge*. On the other hand, *merge* is faster than *proposed* when the data size is small. This is because the partitioning employed by the proposed method generates too small buckets, and, as a result, the massive parallelism of GPUs is not fully utilized.

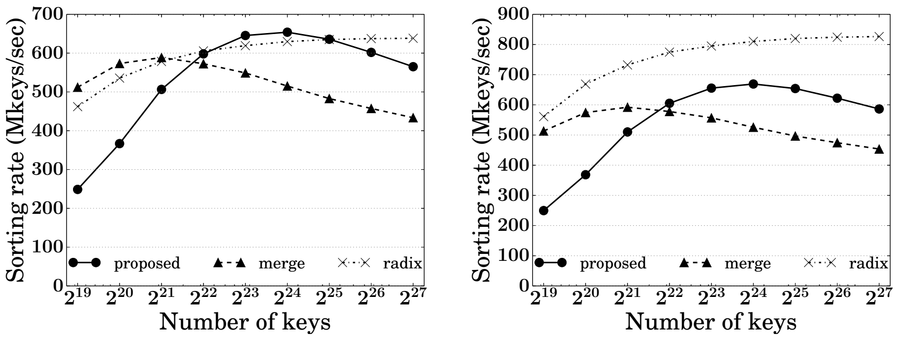
Table 4.1: Time breakdown and the numbers of kernel calls for sorting 2^{25} keys of 32-bit integers.

Kernel	*proposed*		*merge* [8]	
	Time (ms)	# calls	Time (ms)	# calls
Intra-tile sort	16.5	1	13.7	1
Merge	22.4	**6**	56.5	**14**
Splitters	0.8	1		
Bucket sizes	4.4	1		
Scan	0.3	1		
Transposition	0.3	1		
Scatter	8.5	1		
Total	53.2		70.2	

Compared to radix sort, the proposed method shows comparable performance when the number of keys is between 2^{22} and 2^{25}, while radix sort shows superior performance for more larger data. This performance trend is due to the computational complexity of sorting algorithms. Since radix sort is a linear-time algorithm, the performance does not degrade when the data size increases. On the other hand, since merge sort has the complexity of $O(n \log n)$, execution time becomes more than two times longer if data doubles. Consequently, the throughput of merge sort degrades as the number of keys increases. As for the case of 64-bit keys, *proposed* achieved consistently superior performance over *radix*, because the complexity of radix sort depends on the size of keys.

Next we show more detailed comparisons between our method and Baxter sort. Table 4.1 summarizes the number of main kernel calls and kernel's execution time. While Baxter sort required 14 merge passes, the proposed method called the merge kernel 6 times, reducing merge passes by 8. In addition, data partitioning can be carried out in 14.3 ms, which is faster than performing eight merge passes, omitted thanks to partitioning. As a result, overall processing time of our method is 17 ms shorter than that of Baxter sort. Note that intra-tile sort of our method is slightly slower than that of Baxter sort, because the number of blocks for this kernel increases, as mentioned in Section 4.3.2. Also note that, although $k = 128 = 2^7$, the number of merge kernel calls is reduced by eight, larger than seven. This is because the tile size for each implementation is different.

We also conducted experiments using other distributions. Figure 4.7 shows the results with the data generated by normal distribution and exponential distribution.



(a) Normal distribution with mean 2^{30} and standard deviation 2^{26}

(b) Exponential distribution with $\lambda = 10^{-3}$

Figure 4.7: Sorting throughput on different distributions.

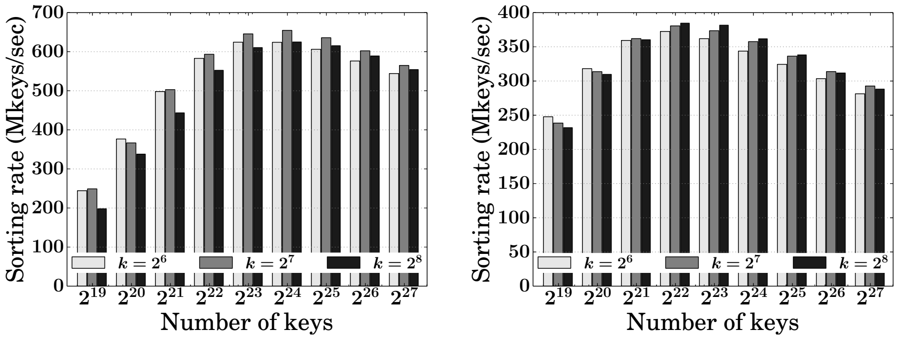
The normal distribution leads to a similar result to the uniform distribution (Figure 4.7(a)). On the other hand, the result of exponential distribution is largely different. While our method and Baxter sort show similar performance to the other distributions, radix sort exhibits better performance. This is because most of the keys have small values. In this case, most of the higher-order bits are 0, and radix sort can exit earlier, thus resulting in higher performance.

4.4.2 Effect of the number of buckets

This section evaluates the effect of the number of buckets on the throughput. Figure 4.8 shows the throughputs on the data generated by a uniform distribution when the bucket number k is set to 2^6, 2^7, and 2^8. Figure 4.8(a) indicates that $k = 2^7$ led to the highest performance for 32-bit integers. On the other hand, for 64-bit integers, either $k = 2^7$ or $k = 2^8$ resulted in the best throughput, depending on the data size: $k = 2^8$ is better for the number of keys between 2^{22} and 2^{25}; $k = 2^7$ is better for the larger number of keys. In the other experiments, we used 2^7 as the value of k because it is more suitable for sorting large-scale data.

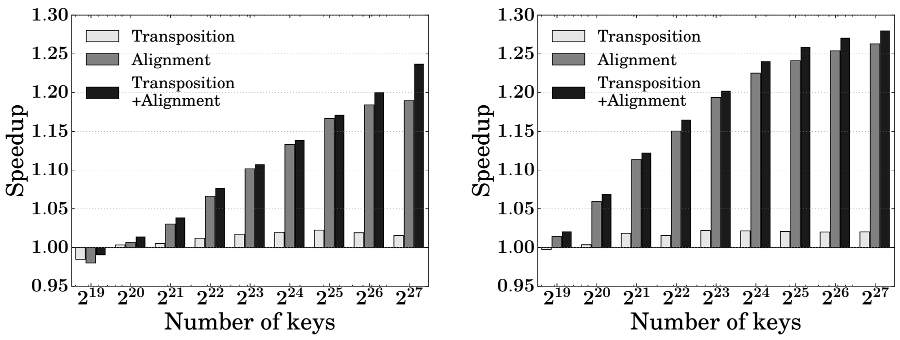
4.4.3 Effect of optimization techniques

This section evaluates the effects of optimization techniques introduced in Section 4.3.1, namely matrix transposition and bucket alignment. Bucket alignment includes the optimization of bucket de-alignment, described in Section 4.3.2. Figure 4.9 illustrates the speedup ratios when the optimizations are applied. In this



(a) 32-bit integers (b) 64-bit integers

Figure 4.8: Throughput with the different numbers of buckets.



(a) 32-bit integers (b) 64-bit integers

Figure 4.9: Speedup ratios by matrix transposition (`Transposition`) and bucket alignment (`Alignment`).

figure, the performance without both of the optimizations is considered as the baseline.

The figures show that bucket alignment is more effective than matrix transposition. The reason is that bucket alignment makes an impact on the whole merge processes, while matrix transposition only affects the scatter step of data partitioning. Since the merge processes account for a large portion of execution time, the alignment turns out to be more effective. Meanwhile, it may become more effective to simultaneously employ both of the optimizations, rather than only using matrix transposition (e.g., the case that the number of keys is 2^{27} in Figure 4.9(a)). The reason of this behavior is considered as follows. Without both of the optimizations, the scatter kernel do not satisfy the condition of coalesced accesses for both read and write operations. Matrix transposition enables coalesced accesses for read operations. However, write operations are still not coalesced accesses. This issue can be mitigated by also adopting bucket alignment. Therefore both reads and writes become coalesced accesses by using the two optimization techniques.

Meanwhile, the increasing numbers of keys make the optimizations more effective. In addition, the speedup ratios of 64-bit integers are higher than those of 32-bit integers. These behaviors are due to increasing required memory bandwidth as the data becomes larger. Thus, the results indicate that the optimization techniques succeeded to enable coalesced accesses. On the other hand, if data is small, the optimizations lead to lower performance. This degradation is due to the overhead caused by the introduction of optimizations.

4.5 Summary

This chapter has described an efficient algorithm for GPU-parallel comparison sorting. The algorithm consists of sample-based data partitioning and cooperative merge. Data partitioning reduces the number of required merges, thereby improving the overall performance. We also introduce two optimization techniques, namely matrix transposition and bucket alignment, to maximize the opportunity of coalesced accesses. Cooperative merge sorts the partitioned buckets with one kernel in a load-balanced manner. Experiments show that, for sufficiently large arrays, our algorithm is superior than the state-of-the-art implementation of merge sort. In addition, for arrays of 32-bit integers generated by a uniform distribution, the proposed algorithm is comparable to radix sort. Furthermore, our algorithm is more than two times faster than radix sort on 64-bit integers.

While this chapter only deals with simple integer data, real-world applications often require sorting of more complex data such as strings. One way to achieve string sorting is to sort the first 32-bit parts of strings and use the other parts of strings if the first 32-bit parts are equal.

Chapter 5

GPU-Accelerated Canopy Clustering

Canopy clustering [90] is a preprocessing method for standard clustering algorithms such as k-means and hierarchical agglomerative clustering [48]. Canopy clustering can greatly reduce the computational cost of clustering algorithms. However, canopy clustering itself may also take a vast amount of time for handling massive data, if we naïvely implement it. To address this problem, we present efficient algorithms and implementations of canopy clustering on GPUs. We not only accelerate the computation of original canopy clustering, but also propose an algorithm using grid index. This algorithm partitions the data into cells to reduce redundant computations as well as to exploit the parallelism of GPUs. Experiments show that the proposed implementations on the GPU is 2 times faster on average than multi-threaded, SIMD implementations on two octa-core CPUs.

5.1 Introduction

Clustering is one of the prominent data mining tasks and has been paid significant attention during the last decade because of the increasing importance of efficient and effective data analytics due to the growing amount of available data. Since clustering algorithms usually require high computational cost, a large number of acceleration techniques have been proposed. One of such techniques is canopy clustering [90], which is a preprocessing method for another clustering algorithm. Canopy clustering first divides a dataset into rougher groups, called canopies, than desired clusters, by using an inexpensive distance measure. Canopy clustering reduces computational cost by restricting distance computations in clustering algorithms only between points within the same canopy. For example, clustering of visual image data is accelerated by canopy clustering [33].

Acceleration of clustering by parallel processing has also been widely studied [26,79,108], and GPUs have recently attracted many researchers and developers for speeding up clustering [15,69,80,118,133,134]. Standard clustering algorithms on GPUs such as k-means [48] have been already well investigated [69, 80, 133]. However, the acceleration of canopy clustering by GPUs has not been attempted. Speeding up canopy clustering on GPUs can lead to the reduction of total processing time of clustering.

This chapter presents efficient algorithms and implementations of canopy clustering for GPUs. First we implement original canopy clustering [90] on GPUs by exploiting efficient implementations of data-parallel primitives [56]. The implementation is further accelerated by a technique called kernel fusion [136], which can decrease the amount of memory accesses. Then we present an algorithm using grid index in order to exploit the spatial locality of data points. The grid index also enables the GPU to efficiently create multiple canopies in parallel. Experiments show that the grid-indexed implementation outperforms a multi-threaded, SIMD implementation on two octa-core CPUs by a factor of 2 on average.

5.2 Preliminaries

Canopy clustering was proposed by McCallum et al. [90] and is employed as a preprocessing step of major clustering algorithms such as k-means and hierarchical agglomerative clustering [48]. Canopy clustering has been paid attention because it can greatly reduce the computational cost of large-scale clustering processing, which takes considerably long time if simply implemented. Canopy clustering firstly divides a dataset into rougher groups, called *canopies*, than desired clusters, by using a simple and inexpensive distance. Having created canopies, it makes more rigorous clusters by using clustering algorithms such as k-means. The main idea of canopy clustering is to reduce unnecessary computations by creating rough groups and restricting distance computations only between points within the same canopy.

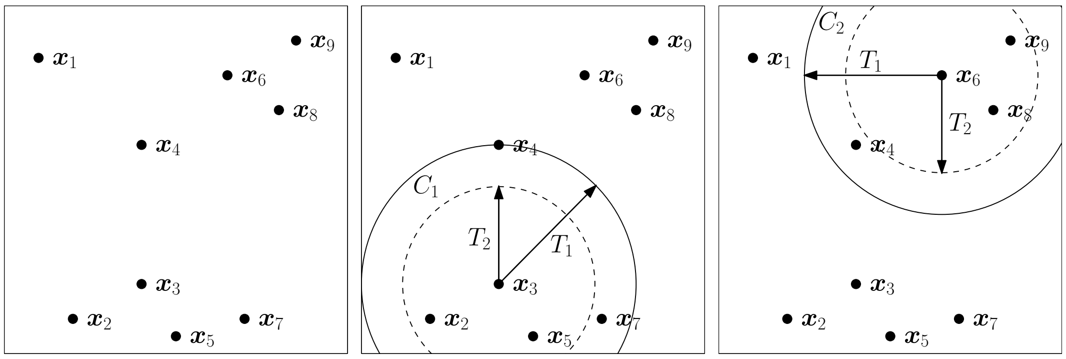
A pseudo-code of canopy clustering is shown in Algorithm 6. There are three inputs: a set of data points and two thresholds T_1 and T_2 ($< T_1$). The first threshold T_1 influences the number of points included in canopies, and T_2 affects the number of canopies created. At first, the algorithm initializes the candidates of centers as the input dataset (Line 2). Then a center point $\boldsymbol{c}$ is picked from the candidates at random (Line 4). A canopy around this center point is to be created in the subsequent steps (Lines 5–13). A canopy includes a data point $\boldsymbol{x}$ if the distance $d(\boldsymbol{x}, \boldsymbol{c})$ between $\boldsymbol{x}$ and $\boldsymbol{c}$ is less than or equal to the threshold T_1 (Lines 7–9). At the same time, the inequality $d(\boldsymbol{x}, \boldsymbol{c}) \leq T_2$ is evaluated and if it holds, the data point $\boldsymbol{x}$ is deleted from

Algorithm 6: Simple Canopy Clustering.

```
Input: A set $S$ of data points $\boldsymbol{x}_i$, thresholds $T_1$ and $T_2$
1  $\mathcal{C} \leftarrow \emptyset$      ▷ $\mathcal{C}$ is a set of canopies
2  $\Sigma \leftarrow S$      ▷ $\Sigma$ is a set of center candidates
3  while $\Sigma \neq \emptyset$ do
4      $\boldsymbol{c} \leftarrow$ get a point from $\Sigma$ at random      ▷ $\boldsymbol{c}$ is a center
5      $C \leftarrow \emptyset$
6      for $\boldsymbol{x} \in S$ do
7          if $d(\boldsymbol{x}, \boldsymbol{c}) \leq T_1$ then
8              $C \leftarrow C \cup \{\boldsymbol{x}\}$      ▷ a canopy $C$ includes a point $\boldsymbol{x}$
9          end
10         if $d(\boldsymbol{x}, \boldsymbol{c}) \leq T_2$ then
11             $\Sigma \leftarrow \Sigma - \{\boldsymbol{x}\}$      ▷ remove $\boldsymbol{x}$ from the candidates
12         end
13     end
14     $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$
15 end
16 return $\mathcal{C}$
```

the center candidates (Lines 10–12). Such canopy creation steps continue until there are no center candidates (Line 3), and finally a set of canopies is returned (Line 16).

Let us consider the example shown in Figure 5.1, where nine 2-dimensional points are plotted. A center $\boldsymbol{c}$ is randomly selected from the nine points, and it turns out to be the point $\boldsymbol{x}_3$ as shown in Figure 5.1(b). A canopy C_1 around this center is created as follows: First, the data points other than $\boldsymbol{c}$ are tested whether their distances against $\boldsymbol{c}$ are less than T_1. If a point satisfy the condition, it is included in the canopy. In the example, the canopy C_1 contains the points $\boldsymbol{x}_2$, $\boldsymbol{x}_4$, $\boldsymbol{x}_5$, and $\boldsymbol{x}_7$. Second, the data points are checked whether they should be deleted from the center candidates. If the distance $d(\boldsymbol{c}, \boldsymbol{x})$ between the center $\boldsymbol{c}$ and a point $\boldsymbol{x}$ is less than or equal to a threshold T_2, this point is excluded from the candidates. Hence the points $\boldsymbol{x}_2$, $\boldsymbol{x}_3$, and $\boldsymbol{x}_5$ are removed from the set of candidates in the example. The next canopy is also constructed in the same way, as shown in Figure 5.1(c), where the point $\boldsymbol{x}_6$ becomes the next center. In this case, the canopy C_2 includes the four points $\boldsymbol{x}_4$, $\boldsymbol{x}_6$, $\boldsymbol{x}_8$, and $\boldsymbol{x}_9$, and the points $\boldsymbol{x}_6$, $\boldsymbol{x}_8$, and $\boldsymbol{x}_9$ are no longer center candidates. The same canopy-creation procedure continues while center candidates exist.



(a) An example dataset. (b) The first canopy. (c) The second canopy.

Figure 5.1: An example of canopy clustering.

```
while there exist candidates do
    compute_distances();
    create_canopy();
    select_center();
end
```

Figure 5.2: GPU implementation of the while loop in Algorithm 6.

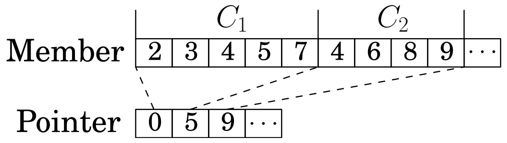


Figure 5.3: Two arrays representing canopies.

5.3 GPU parallelization

We divide the iteration of Algorithm 6 into three kernels: `select_center`, `compute_distances`, and `create_canopy`. In addition, for the convenience of checking termination condition, we slightly change the `while` loop as shown in Figure 5.2. That is, centers are selected at the last of iterations rather than the first.

The following sections explain data structures for easing efficient implementations and then describe implementations of the three kernels. We also introduce an optimization technique called *kernel fusion*.

5.3.1 Data structures

We need to maintain the following data on global memory: input data points, resulting canopies, and the information of center candidates.

The input data is a set of n d-dimensional data points, which can be represented by an $n \times d$ matrix. This data is read-only and is accessed only when distances are computed. To enable coalesced accesses in distance computations, the matrix is stored in a column-major format.

Resulting canopies are stored as two 1-dimensional arrays in a similar way to sparse-matrix formats [10]. Figure 5.3 shows the two arrays with regard to the previous example (Figure 5.1). The member array holds which data points belong to canopies, while the index array maintains where the boundaries of canopies lie in the member array. For example, the members of canopy C_1 are stored in the member array at indices 0 through 4.

The information of center candidates is maintained as an integer array of n elements. The ith element indicates whether the ith data point is currently a candidate or not. If the ith data point is still a candidate, we store $i - 1$ as the ith element; otherwise n is assigned to indicate that it is no longer a candidate. This format enables an easy and sufficiently efficient implementation of center selection, which is described next.

5.3.2 Selecting centers

Before constructing a canopy, we need to select a center from the set of center candidates. To this end, we decide to choose the data point of minimum index among candidates as a center, for the ease of implementation. Intuitively this selection policy is not significantly different from random selection under the assumption that the data points are given in random order.

The selection policy can be easily implemented with the data structure of center candidates by using the reduce primitive. If we use the `min` operation as the binary operator of reduce and apply it to the array of center candidates, the above selection policy is realized. If the result of reduce is less than n, then there still exist candidates; otherwise there is no more candidate and thus the execution should be terminated.

5.3.3 Computing distances

Having selected a center, we compute distances between the center and points. In this work, we focus on the Euclidean distance although other distances are also applicable. Paralellization of distance computations is realized by simply assigning one thread to one distance computation. Since the data matrix on global memory is stored in column-major order, the threads can access the data points in a coalesced manner, thereby achieving high effective memory bandwidth. Having finished the distance computations, threads test the distances against the two thresholds T_1 and T_2. The threads store the information whether the distance is less than or equal to T_1

into a T_1-flag array: The ith element of this array equals 1 if $d(\boldsymbol{c}, \boldsymbol{x}_i) \le T_1$; otherwise the element is 0. Meanwhile, if the distance is less than or equal to T_2, the candidate is excluded from the set of center candidates, by assigning n to the corresponding position of center-candidate array.

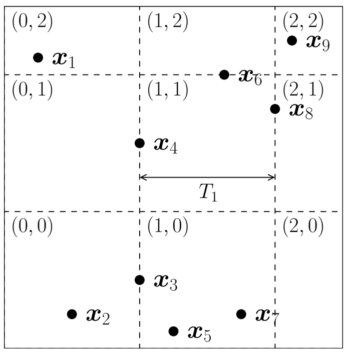
5.3.4 Creating canopies

This step gathers the indices of data points included in the canopy, on the basis of the result of the last step. To this end, this step uses the filter primitive, described in Section 2.1.2. More specifically, the input array to filter is the T_1-flag array and the predicate P is the function that returns true when the ith element of T_1-flag array equals 1.

5.3.5 Kernel fusion

We here introduce an optimization technique called *kernel fusion* [136]. Kernel fusion is, as the name indicates, to combine multiple kernels into a single kernel, thereby reducing the number of kernel calls and achieving better utilization of resources. The above-explained kernels perform few computations and thus they are bandwidth-bound kernels. By introducing kernel fusion, we can expect performance improvements because it typically reduces the amount of accesses to global memory.

We combine the three kernels (i.e., `compute_distances`, `create_canopy`, and `select_centers`). In one kernel, a block deals with a chunk of input data, computes distances of the points in the chunk against the center, creates a part of canopy, and selects the next center. Having finished the distance computations, a block creates a part of canopy on shared memory, and adds the number of canopy members to a counter on global memory, which is initialized to 0 in advance. For this addition, we use an atomic function, which returns the original value of target variable. The returned value can be used to the output index of canopy-member array. Thereby blocks write out canopy members to global memory in parallel. Meanwhile, center selection is done by using an atomic `min` function: First a block-level candidate is selected on shared memory by the atomic function, and then one thread in a block updates the candidate information on global memory, which is initialized to n. One important point to note is that the canopy members are not properly sorted with the above scheme because of the atomic function. Thus, we sort the canopy members after all the canopies are generated, by using segmented sort, which sorts multiples segments in one array on a segment-by-segment basis.



(a) Partitioned data space.

$$
\begin{aligned}
&G_{0,0} = \{\boldsymbol{x}_2\},\ \mathcal{N}(G_{0,0}) = \{G_{1,0}, G_{1,1}\}\\
&G_{0,2} = \{\boldsymbol{x}_1\},\ \mathcal{N}(G_{0,2}) = \{G_{1,1}, G_{1,2}\}\\
&G_{1,0} = \{\boldsymbol{x}_3, \boldsymbol{x}_5, \boldsymbol{x}_7\},\ \mathcal{N}(G_{1,0}) = \{G_{0,0}, G_{1,1}, G_{2,1}\}\\
&G_{1,1} = \{\boldsymbol{x}_4\},\\
&\quad \mathcal{N}(G_{1,1}) = \{G_{0,0}, G_{0,2}, G_{1,0}, G_{1,2}, G_{2,1}, G_{2,2}\}\\
&G_{1,2} = \{\boldsymbol{x}_6\},\ \mathcal{N}(G_{1,2}) = \{G_{0,2}, G_{1,1}, G_{2,1}, G_{2,2}\}\\
&G_{2,1} = \{\boldsymbol{x}_8\},\ \mathcal{N}(G_{2,1}) = \{G_{1,0}, G_{1,1}, G_{1,2}, G_{2,2}\}\\
&G_{2,2} = \{\boldsymbol{x}_9\},\ \mathcal{N}(G_{2,2}) = \{G_{1,1}, G_{1,2}, G_{2,1}\}
\end{aligned}
$$

(b) Corresponding grid cells.

Figure 5.4: An example of grid index.

5.4 Canopy clustering with grid index

The simple canopy clustering algorithm has the disadvantage that it performs a significant number of redundant computations, because it does not make use of spatial locality of data points. This section introduces an accelerated algorithm of canopy clustering by using a grid index structure. Section 5.4.1 describes the grid index, and then Section 5.4.2 explains implementation details.

5.4.1 Grid index

The grid index is constructed by partitioning the data space into equi-width hypercubes, which we call *grid cells*. A grid cell $G_{i_1,i_2,\dots,i_d}$ in d-dimensional space can be defined as a set $\{\boldsymbol{x} \in S \mid i_j \cdot w \leq x_j < (i_j + 1) \cdot w, 1 \leq j \leq d\}$, where S is a set of data points and w is the cell width. The subscript $(i_1, i_2, \dots, i_d)$ is called *cell coordinates*. In our method, a threshold T_1 is chosen as the width w of grid cells, so that canopy creation is accomplished by computing the distances only between data points in a cell and its neighbor cells. A cell is called a *neighbor* of another cell if each dimension of the cell coordinates does not differ more than one, and the information of neighbors is also associated to grid cells. The other cells do not need to be taken into consideration in canopy creation because if two cells are not neighbors, the distance between two points in each cell must be greater than T_1.

Figure 5.4 shows an example of partitioned data space and the corresponding grid cells. In Figure 5.4(b), $\mathcal{N}(G)$ means a set of neighbors of G. By using the grid index, we can create a canopy as follows: Let us consider that the point $\boldsymbol{x}_3$

is selected as a center. Since this point belongs to the cell $G_{1,0}$, the distances are computed against the points in $G_{1,0}$ and its neighbor cells. In this case, the neighbors are $G_{0,0}$, $G_{1,1}$, and $G_{2,1}$. Distances are computed against five points (i.e., $\boldsymbol{x}_2$, $\boldsymbol{x}_4$, $\boldsymbol{x}_5$, $\boldsymbol{x}_7$, and $\boldsymbol{x}_8$). Thus we can omit the distance computations with regard to the other three points (i.e., $\boldsymbol{x}_1$, $\boldsymbol{x}_6$, and $\boldsymbol{x}_9$).

5.4.2 Implementation

This section presents how to efficiently implement canopy clustering with grid index on GPUs. It is especially important to efficiently construct a grid index on GPUs and to create multiple canopies in parallel. The following explains how to construct a grid index, how to select multiple centers, and how to create multiple canopies in parallel, one by one.

Constructing grid index on the GPU

A grid index can be represented by two sparse binary matrices: a matrix of cell members and a matrix of neighbor information. A sparse binary matrix, in turn, can be expressed by two 1-dimensional arrays as used for canopies. Thus we store a grid index on the GPU as four 1-dimensional arrays in total. Such a grid index is efficiently constructed on the GPU by exploiting fast data-parallel primitives in three steps: (1) computing cell coordinates from data points and sorting them; (2) building the cell-member matrix on the basis of the coordinates; and (3) constructing the neighbor matrix on the basis of the coordinates.

Computing and sorting cell coordinates. The first step computes the cell coordinates of each data point (i.e., the coordinates of the cell to which the point should belong), and these coordinates are stored on global memory in column-major order. Then the coordinates are sorted in the dictionary order with data-point indexes as associated data. On the basis of an example of Thrust[1], we realize the dictionary-order sort by sorting dimension by dimension from the last: If there are d dimensions, first the dth dimension with data indexes is the target of sorting, and then the $(d-1)$th dimension is subject to next sorting. Before the sorting of $(d-1)$th dimension, the coordinates of $(d-1)$th dimension is not consistent with the order of dth dimension. Thus it is necessary to arrange the order according to the data-index array. Having arranged the coordinates, we sort the $(d-1)$th dimension. The sorting procedure continues until the sorting of first dimension completes. Finally, the dimensions other than the first need to be arranged in the final dictionary order. As for sorting,

[1]`https://code.google.com/p/thrust/`

we used the sort function of Thrust library. We also used the gather function of Thrust for arranging the dimensions according to the data indices.

Building the cell-member matrix. Having sorted the coordinates, we next construct the cell-member matrix. The array of cell members is already obtained at this time, because it is the array of data indexes sorted in the first step. Thus it is sufficient to compute the array of pointers in order to finish the construction of cell-member matrix. This array can be easily obtained by examining the sorted coordinates: If the coordinates of a point are different from the coordinates of the next point, the index of the next point indicates the boundary of same cell members.

This step can be easily parallelized on the GPU by assigning threads to points. A thread checks the coordinates of the assigned point and the next point, and stores the index to an array on global memory if the two points have different coordinates. For enabling concurrent writes to the resulting array, we prepare a global counter of current index for the pointer array. This counter is first set to 1, because we initialize the first element of pointer array to 0. If a thread wants to write an index to the pointer array, the thread increments the counter by 1 with an atomic function. By using the value returned from the atomic function as the output index of pointer array, the thread writes the boundary pointer value. Since we use an atomic function, the write order is not deterministic. This means that the elements in the pointer array is not properly ordered. Thus we sort the pointer array so that adjacent elements of this array stands for the boundaries of cell members.

Constructing the cell-neighbor matrix. The last step is to construct a cell-neighbor matrix. To this end, firstly the unique cell coordinates are gathered from the sorted coordinates created in the first step. With the coordinates, this step is performed in two passes: The first pass just counts the number of neighbors of each cell, and the second pass constructs a cell-neighbor matrix according to this information.

The first pass simply checks all of the combinations of coordinates. This is parallelized by assigning one block to the coordinates of one cell. Within the block, one thread compares the assigned coordinates and coordinates of another cell. Each block outputs the counting result to a corresponding location of pre-allocated array on global memory. Then exclusive scan is performed over this array, and the output becomes the pointer array of neighbor matrix.

The second pass similarly checks all of the combinations, but this time outputs cell-neighbor indices to global memory. Since we have already obtained the pointer array, it is known where a block should output the neighbor indices. In order to enable threads to write concurrently, a counter is used again. The counter is prepared on shared memory and is initialized to 0. If a thread have found a neighbor, the counter is incremented and the neighbor index is stored to the array on global memory.

Selecting multiple centers

The selection of multiple centers is performed in two phases: selection and verification phases. The selection phase chooses a canopy center per grid cell, so that there are canopy centers as many as the grid cells. The verification phase checks whether or not the selected centers violate a condition derived from the canopy-creation process: The distance of two canopy centers is not less than a threshold T_2. If a pair of centers violates this condition, a point of the pair is invalidated as canopy center at current iteration.

The selection phase selects centers by reduction in a similar way as in Section 5.3.2. More specifically, one block deals with one cell and selects a center from the points in the cell. A block performs the reduction on shared memory and one thread of the block outputs the selected center to a pre-allocated array of as many elements as the number of cells. Note that the information of candidates is maintained on global memory as in Section 5.3.2.

The verification phase checks the above-described condition of centers. In order to check multiple centers in parallel, we employ a rather pessimistic scheme: A center is invalidated if one of the other centers is closer than the threshold T_2, although the other centers may be invalidated by other blocks concurrently. To realize this scheme, one block checks the validity of one center, and one thread within the block compares the center against another center. This scheme may over-invalidate centers, but experiments in Section 5.5 indicate that it can provide a sufficient level of parallelism.

Creating multiple canopies

In order to generate multiple canopies in parallel, while basically the same procedure to simple canopy clustering can be employed, the difficulty lies in how to assign the work to thread blocks for the creation of canopies. A naïve scheme is to make one canopy by one block, but it could suffer from severe load imbalance, because the required number of distance computations largely varies canopy by canopy. To alleviate this problem, we assign one block to either a cell that contains a center or one of the neighbors of such a cell. Blocks compute distances between a center and the points of assigned cell, and generate a part of canopy. For example, let us consider to create a canopy around the point $\boldsymbol{x}_3$ in Figure 5.4. In this case, one block handles the cell $G_{1,0}$ and other three blocks are responsible for the neighbor cells $G_{0,0}$, $G_{1,1}$, and $G_{2,1}$ with respect to the cell $G_{1,0}$.

5.5 Experiments

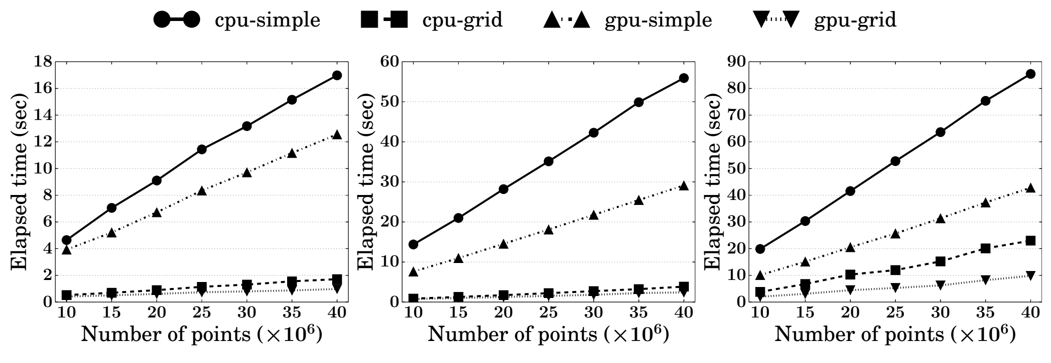
This section evaluates the performance of our proposed implementations. Section 5.5.1 summarizes the experimental settings and Section 5.5.2 compares CPU and GPU implementations with and without grid index.

5.5.1 Experimental setup

We used CUDA 6.5 for GPU implementations and conducted experiments on an NVIDIA Tesla K40 GPU, which has global memory of 12 GB and 15 SMs, each of which consists of 192 SPs at 745 MHz. The machine has two Intel Xeon Processor E5-2687W v2 at 3.40 GHz with memory of 128 GB. The OS is CentOS release 6.5 (Final).

We compared four implementations: `cpu-simple`, a CPU implementation of canopy clustering; `cpu-grid`, a CPU implementation with grid index; `gpu-simple`, a GPU implementation of canopy clustering including kernel fusion; and `gpu-grid`, a GPU implementation with grid index. We parallelized the CPU implementations by ourselves using OpenMP and AVX, which is SIMD functionality on Intel CPUs [85]. The multi-threaded, SIMD implementations on two octa-core CPUs were up to 17 times faster than serial implementations.

We used synthetic datasets for evaluation because we had difficulty in finding suitable real-world datasets. The synthetic datasets are generated as follows: The number of clusters is given as an input, and each cluster is represented by a tuple of d normal distributions, which are independent with each other. The means of normal distributions are generated by the uniform distribution between 0 and 1, and the standard deviations are given by the normal distribution with mean 0.01 and standard deviation 0.005. With these clusters, the given number of points are generated one by one: First we determine which cluster a new point belongs to in a uniformly-random manner, and then the point's coordinates are given by the normal distributions of the cluster. We set the number of clusters to 100 in the experiments. The dimensionality d was varied from 2 to 6; this may seem small, but canopies are often constructed on the basis of a small number of attributes [90]. Thus the dimensionality is not very small for canopy clustering.



(a) $d = 2, T_1 = 0.02$. (b) $d = 4, T_1 = 0.15$. (c) $d = 6, T_1 = 0.35$.

Figure 5.5: Elapsed times with varying sizes of synthetic datasets.

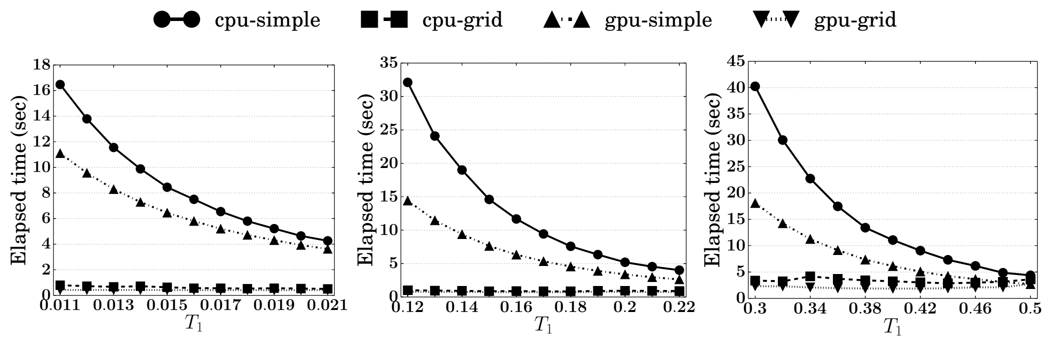
5.5.2 Results

Varying dataset sizes

Figure 5.5 shows the elapsed times of the four implementations with varying sizes of datasets. Note that the elapsed time of GPU implementations includes CPU–GPU data-transfer time, but does not include GPU–CPU data-transfer time because we assume that the result of canopy clustering is used in later clustering on GPUs. For this experiment, the threshold T_1 is chosen such that the number of canopies become around 3,000, and the other threshold T_2 is fixed to $0.7T_1$ for the experiment purpose; in practice, the thresholds should be determined according to the needs of applications.

Overall, each kind of GPU implementations is consistently faster than the corresponding CPU implementation. The `gpu-simple` implementation outperforms `cpu-simple` by a factor of 1.4 on two-dimensional datasets, 1.9 on four-dimensional datasets, and 2.0 on six-dimensional datasets. As the dimensionality increases, the GPU implementation gets faster than the CPU implementation. This is because canopy clustering of higher-dimensional datasets results in more bandwidth-bound kernels. Consequently the wide memory bandwidth of GPUs become more effective in higher-dimensional cases.

The GPU implementation using grid index (i.e., `gpu-grid`) is also faster than `cpu-grid`, leading to speedup of 1.3–1.8 in two dimensions, 1.1–1.6 in four dimensions, and 1.9–2.5 in six dimensions. From this result, we believe that our scheme of multiple center selection (Section 5.4.2) achieves to provide sufficient level of parallelism for GPUs. However, `gpu-grid` becomes less efficient in four-dimensional



(a) $d = 2$. (b) $d = 4$. (c) $d = 6$.

Figure 5.6: Elapsed times with varying the thresholds T_1 and T_2.

case. This can be considered due to load imbalance: Since we assign one block to one grid cell, the processing time of blocks varies depending on the size of grid cells, which can be largely different by dataset characteristics and thresholds. This load-imbalance problem should be remedied in our future implementations.

Varying thresholds

We here fix the size of datasets to 10 million and change the thresholds T_1 and T_2, which affect the number of canopy members and the number of canopies, respectively. Figure 5.6 shows the elapsed times of the four implementations. The value of T_1 is varied so that the number of canopies lies in roughly the range of 1,000 and 10,000. The threshold T_2 is fixed to $0.7T_1$ as before.

The implementations using grid index get faster and faster as the thresholds decrease, compared to the implementations without grid index. This is because more canopies are generated when thresholds are smaller. The implementations with grid index carry out distance computations between a center and a limited number of points, while the implementations without grid index perform distance computations between a center and all of the other points at each iteration. In other words, the more canopies are generated, the more efficient the implementation using grid indices are, compared to the simple canopy clustering. Meanwhile, in six-dimensional case (Figure 5.6(c)), the speedup ratios between `gpu-grid` and `cpu-grid` range from 1.33 to 2.01. Such instability comes from the load-imbalance problem as before, because the organization of grid index differs according to the thresholds,

5.6 Summary

This chapter has presented GPU implementations of canopy clustering, parallelizing original canopy clustering and canopy clustering with grid index. The implementation of canopy clustering is accelerated by the kernel-fusion technique. Canopy clustering using grid index exploits the spatial locality of input data points, thereby reducing the number of distance computations and providing high parallelism to the GPU. Experiments showed that the kernel-fused implementation and the GPU implementation with grid index outperform multi-threaded, SIMD implementations on two octa-core CPUs by a factor of up to 2.

Chapter 6

GPU-Accelerated Graph Clustering

Graph clustering has recently attracted much attention as a technique to extract community structures from various kinds of graph data [35]. However, since available graph data becomes increasingly gigantic, the acceleration of graph clustering is an important issue for handling large-scale graphs. This chapter describes a fast graph clustering method using GPUs. The proposed method is based on one of the fastest graph clustering algorithms, label propagation [112]. To efficiently process graph clustering, the algorithm of label propagation is firstly transformed into a sequence of data-parallel primitives. In addition, load balancing is taken into account by using the primitives that make the load among threads and blocks well balanced. Experiments on both real-world and synthetic datasets show that our proposed method is superior than existing methods, in terms of not only efficiency but also accuracy.

6.1 Introduction

Graph clustering (also known as community detection) is a technique to extract community structures, or clusters, from graph-structured data, such that vertices in one cluster are densely connected and vertices in different clusters are sparsely connected [35]. Graph clustering has been successfully applied for various kinds of graph data ranging from the Web and online social networks to biological data [106]. However, recent graph data becomes increasingly gigantic, and it is difficult to handle such large-scale data by existing methods in realistic time. Thus it is highly important to accelerate the processing of graph clustering.

Among existing algorithms, label propagation is known as one of the fastest algorithms [112]. It has the computational complexity linear to the number of edges, which is faster than many other algorithms. In addition, it is reported that the clustering accuracy of label propagation is also better than state-of-the-art algorithms [131]. Furthermore, label propagation is suitable for parallel processing [78], because it makes clusters by only using vertex-local (i.e., adjacent vertices) information. Thus label propagation on GPUs is considered as a promising solution for accelerating graph clustering.

There exist several challenges to harness the power of GPUs. Among them, load balancing is a particularly important issue, given the scale-free property of real-world graphs and the massive parallelism of GPUs. A scale-free graph is a graph whose degree distribution follows a power law [7]. This means that most vertices have low degrees, while few vertices have extremely high degrees. Therefore, if we naïvely parallelize the algorithm by assigning the workload of one vertex to one thread, the implementation suffers from heavy load imbalance. Label propagation using GPUs has already been proposed by Soman and Narang [121]. However, their method has the limitation that it does not well take into account load balancing.

In this chapter, we present a GPU-based method to accelerate graph clustering by parallelizing label propagation with achieving load balancing. To efficiently execute label propagation on GPUs, the label-propagation algorithm is firstly transformed into a sequence of data-parallel primitives. In addition, load balancing is taken into account by using the primitives that make the load among threads and blocks balanced, even if the skew of degrees exists. Experiments on both real-world and synthetic datasets show that our proposed method is superior than existing methods.

6.2 Preliminaries

This section describes the problem of graph clustering in Section 6.2.1 and label propagation in Section 6.2.2.

6.2.1 Problem definition

Let $G = (V, E)$ be a graph, where V is the set of vertices and $E \subset V \times V$ is the set of edges. Unless otherwise noted, we denote the numbers of vertices and edges as n and m, respectively. For the sake of simplicity, this work assumes that graphs are undirected and unweighted, but the proposed method can be easily extended to handle directed and weighted graphs. The problem tackled in this chapter is defined as follows:

Algorithm 7: Label propagation

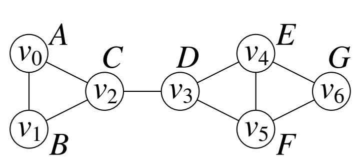
```
   Input: A graph $G = (V, E)$
   Output: a set $C$ of clusters
1  Initialize label assignment $L$        ▷ $L_i$ denotes the label of a vertex $v_i$
2  repeat
3      $\pi \leftarrow$ a random permutation of the sequence $(1, 2, \ldots, n)$
4      for $i \in \pi$ do
5          $L_i \leftarrow \arg\max_{l \in L} \left|\{j \mid (v_i, v_j) \in E \wedge L_j = l\}\right|$
6      end
7  until a termination condition is met
8  $C \leftarrow$ extract clusters
9  return $C$
```

Problem 2 (Graph clustering)**.** *Given a graph $G = (V, E)$, find a set of clusters, $C = \{C_1, C_2, \ldots, C_l\}$, that satisfy the following conditions:* (1) *for any i, $C_i \subset V$;* (2) *for any i and j ($i \neq j$), $C_i \cap C_j = \emptyset$;* (3) $\bigcup_{i=1}^{l} C_i = V$*; and* (4) *vertices in the same cluster are densely connected, while vertices between different clusters are sparsely connected.*

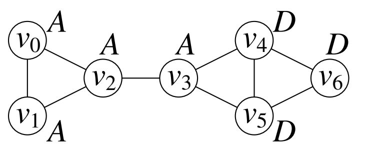
6.2.2 Label propagation

Label propagation, proposed by Raghavan et al. [112], is one of the fastest graph-clustering algorithms. It clusters a graph on the basis of the idea that a vertex v belongs to the cluster to which the majority of the adjacent vertices of v belongs. Algorithm 7 shows a pseudo-code of label propagation. First, vertices are initialized by unique cluster labels (Line 1). Then the labels of vertices are to be updated in random order (Lines 3–6). A label of a vertex v_i is updated to the label that most frequently appears among the adjacent vertices of v_i (Line 5). If multiple labels appear the same number of times, the tie is broken randomly. This update iteration continues until some termination condition is met, and vertices with the same label are extracted as elements of the same cluster (Lines 8–9). Several termination conditions can be considered, such that the number of updated label in one iteration becomes sufficiently small, or the number of iterations exceeds a threshold.

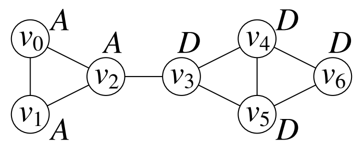
Figure 6.1 illustrates an example of label propagation. First, vertices are associated with unique labels (Figure 6.1(a)). The labels are to be updated in random order of vertices. In this example, let the order be $(v_2, v_4, v_6, v_1, v_3, v_0, v_5)$. The first vertex v_2 has three adjacent vertices v_0, v_1, and v_3 whose labels are A, B, and D,



(a) The initial state.



(b) Labels have been updated in the order $(v_2, v_4, v_6, v_1, v_3, v_0, v_5)$.



(c) The steady state.

Figure 6.1: Example of label propagation: the letters near vertices denote the labels.

respectively. In this case, all of the labels appear the same number of times. Thus the label of v_2 is randomly updated to one of the three labels, and it turns out to be A. The label of the next vertices are similarly updated. Figure 6.1(b) shows the situation that all the labels are updated once. Subsequently, when all vertices are iterated once again (Figure 6.1(c)), the labels become fixed (i.e., labels are no longer updated). Finally, two clusters, $C_A = \{v_0, v_1, v_2\}$ and $C_D = \{v_3, v_4, v_5, v_6\}$, are obtained as the clustering result.

6.3 Proposed method

This section describes the proposed method, GPU-accelerated label propagation. In label propagation, the labels of adjacent vertices (or *adjacent labels* for short) of a vertex need to be accumulated, and the majority label needs to be computed for updating the label. The proposed method executes this computation efficiently on the GPU by exploiting multiple data-parallel primitives. Meanwhile, if we simply assign a thread block to a vertex, the performance significantly degrades because of load imbalance, given the scale-free property of real-world graphs. Thus, the proposed method parallelizes the computation by utilizing *skew-resilient* operators, whose performance is not affected by the existence of skewness. In the rest of this section, Section 6.3.1 introduces the data layout on the GPU. Section 6.3.2 describes the approach for counting adjacent labels, and Section 6.3.3 discusses the label updating scheme.

6.3.1 Data layout

To execute label propagation, the following data needs to be maintained on the GPU: (1) input graph data and (2) vertex labels. Our method represents the input graph data by the *CSR (compressed sparse row)* format, which is a format for sparse

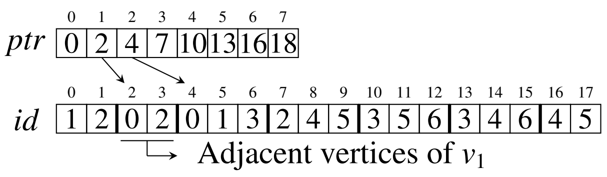


Figure 6.2: Graph data on the GPU. A graph is represented by two arrays *id* and *ptr*.

matrices [10], because a graph can be regarded as a matrix and it is usually sparse. The advantages of the CSR format, compared to other sparse-matrix formats, are low space cost and its aptitude for data-parallel primitives. Meanwhile, the label information is simply stored as an array of integers where an element is the label of a corresponding vertex. Note that, although the examples of this chapter denote the labels as letters for the sake of explanation, labels are stored as integers in the implementation.

Figure 6.2 illustrates an example of graph data by the CSR format. A graph is represented by two arrays *id* and *ptr*. The array *id* contains the indices of adjacent vertices. The array *ptr* maintains the offsets where the indices of adjacent vertices for a specific vertex start in the array *id*. For example, the information of the adjacent vertices of v_1 in Figure 6.1 is stored as the elements of the array *id* from $ptr[1] = 2$ to $ptr[2] = 4$. In general, the CSR format represents a sparse matrix with one more array to store the value of each entry. However, since entries of adjacent matrices of unweighted graphs are 0 or 1, such an array is omitted in this case.

6.3.2 Counting adjacent labels

This section describes an approach for counting adjacent labels, which consists of the following four steps: (1) gathering adjacent labels, (2) sorting the adjacent labels per vertex, (3) extracting boundaries of the adjacent labels, and (4) computing the occurrence frequencies of adjacent labels. This approach is based on the following idea. If the adjacent labels of each vertex is sorted, the same adjacent label of one vertex is stored contiguously. This means that, if we have the indices where the adjacent labels change, the occurrence frequencies can be easily computed by the differences of such indices (or *boundaries*), without a specific data structure such as an associative array. In other words, the counting can be implemented with simple operations on arrays, which are suitable for GPUs. The following details the four steps.

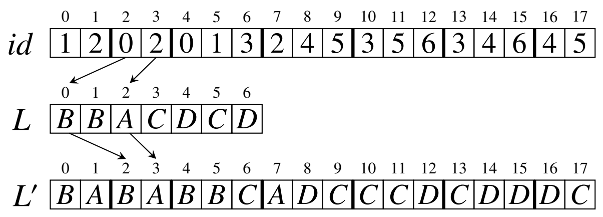


Figure 6.3: Gathering adjacent labels. The *i*th element of the array L denotes the label of vertex v_i.

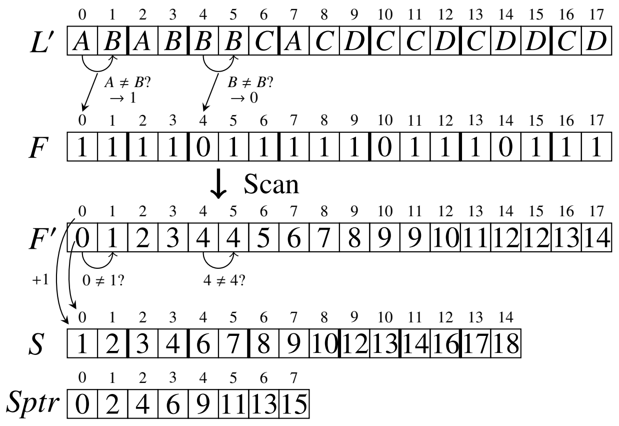


Figure 6.4: Extracting boundaries.

Gathering adjacent labels

The first step creates an array L' of adjacent labels by using the adjacent-vertex array *id* and the label array L. Figure 6.3 shows an example of this step. This step can be implemented by using the gather primitive, by passing the two arrays as input.

Sorting adjacent labels per vertex

This step sorts the array L' of adjacent labels in a per-vertex manner. This means that each subarray of L' regarding one vertex is separately sorted. A naïve way is to sort one subarray by one block. However, as previously mentioned, this scheme suffers from severe load imbalance because of the skewness of degree distributions. Instead, we use a data-parallel primitive, segmented sort, whose performance is not largely affected by the skewness. Specifically, the arrays L' and *ptr* are passed to segmented sort as input.

Extracting boundaries

The third step extracts the boundaries of adjacent labels from the sorted array L' in the previous step. This step is further divided into three sub-steps: (1) checking the boundaries of adjacent labels, (2) computing offsets for creating boundary arrays, and (3) extracting the boundaries. Figure 6.4 illustrates an example of these sub-steps. The goal here is to construct the two arrays S and *Sptr*. These arrays play a similar role to arrays in the CSR format: the first array S contains the information of boundaries, which are the indices where labels change, while the second array *Sptr* maintains the offsets where the boundaries for a specific vertex begin in the array S.

The first sub-step creates an array F of flags to indicate the indices where adjacent labels change. This computation is parallelized in a similar way to the map primitive. The ith thread takes the ith and $(i + 1)$th elements of L', and judges whether the two labels are different or not. If they are different, the thread stores '1' to $F[i]$; otherwise '0' is stored to $F[i]$. Meanwhile, even if the two labels are same, adjacent labels from different vertices are considered different. Thus '1' is stored to F in such cases. For example, although $L'[3] = L'[4]$ in Figure 6.4, $F[3]$ is computed as '1' because $L'[3]$ is an adjacent label of v_1 and $L'[4]$ is an adjacent label of v_2.

The next sub-step computes offsets for creating boundary arrays in the next sub-step. This computation can be easily implemented by using the scan primitive. The inputs are the array F, the addition operator +, and 0 as the identity element. The array F' in Figure 6.4 is the result of this primitive.

With the offsets, the arrays S and *Sptr* are constructed. Similarly to the first sub-step, the ith thread compares the ith and $(i + 1)$th elements of F'. If they are different, the thread output boundary information; otherwise, it just exits without outputs. When outputting, the ith thread writes $i + 1$ to the array S at the index of $F'[i]$. That is, the thread executes $S[F'[i]] \leftarrow i$. Meanwhile, the array *Sptr* is also computed to maintain the correspondence between boundaries and vertices. This array is computed by executing $Sptr[i] \leftarrow F'[ptr[i]]$ in parallel.

Computing the occurrence frequencies

This step calculates the array W of occurrence frequencies from the array S. Each element of W is computed by one thread: An element of W is the difference between two adjacent elements of S; i.e., $W[i] \leftarrow S[i+1] - S[i]$. For the boundary case (i.e., the computation of $W[0]$), $S[i]$ becomes directly the frequency.

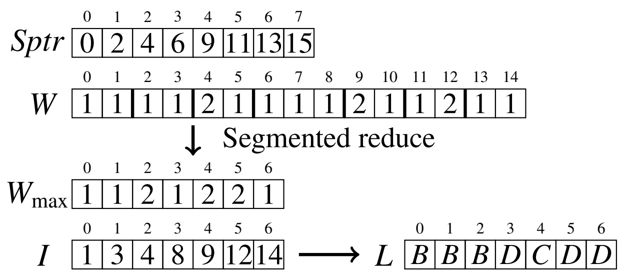


Figure 6.5: Updating labels.

6.3.3 Updating labels

By using the results obtained by the method in Section 6.3.2, vertex labels are updated on the GPU. To this end, we need to compute the majority of adjacent labels per vertex, from the array W of occurrence frequencies. This computation can be implemented by using the reduce primitive. Reduce with the max operator, which takes two operands and returns the larger one, is applied to each subarray of W in a per-vertex manner. Then the result is the maximum frequency for each vertex. However, this parallelization may suffer from severe load imbalance because of the skewness of degree distributions, as previously mentioned. To alleviate this issue, the proposed approach employs the segmented reduce primitive, which is able to efficiently process skewed data. Originally, segmented reduce only returns the array of reduced values (i.e., frequencies), while we need the labels with the maximum frequencies for updating labels. Thus, we extend the primitive so that it also returns an array I of indices corresponding to the reduced values. By using this array, we can update the labels by computing $L[i] \leftarrow L'[S[I[i]]]$ in parallel. Figure 6.5 shows an example of this step.

6.4 Experiments

This section evaluates the performance of the proposed method through experiments. We used a server running CentOS release 6.5 (Final) as the OS, equipped with Intel Xeon Processor E5-2687W v2 at 3.40 GHz and 128 GB memory. The GPU is NVIDIA Tesla K40, which includes 15 SMs, 2,880 SPs, and 12 GB.

The proposed method (*GPU-LP*) was implemented using CUDA 7.0. For comparison, three implementations were used:

- *Louvain* [14]: this is a famous algorithm based on maximizing modular-

Table 6.1: Real-world datasets.

Dataset	n	m
amazon	334,863	925,872
dblp	317,080	1,049,866
youtube	1,134,890	2,987,624
livejournal	3,997,962	34,681,189
orkut	3,072,441	117,185,083

ity [97], which is a measure to evaluate clustering quality. It is widely known that this algorithm produces accurate results in terms of modularity with short processing time. An implementation is made publicly available by the original authors. Thus we used this implementation as a kind of baseline method, although *Louvain* generates different kinds of clustering results from label propagation.

- *CPU-LP*: a serial CPU implementation of label propagation, implemented by ourselves.
- *GPU-LILP* (load-imbalanced label propagation): a GPU-based implementation does not take into account load balancing. This implementation is different from *GPU-LP* in that *GPU-LILP* does not use the segmented sort and segmented reduce primitives. Instead, *GPU-LILP* parallelize these computations in a block-per-vertex manner. As previously mentioned, this parallelization may suffer from severe load imbalance. This implementation was included to evaluate the impact of load balancing.

The termination condition for label propagation was set as the number of updated labels in one iteration is less than $10^{-5}n$ or the number of iterations reaches ten, where n is the number of vertices.

In the following, we first evaluate the performance on real-world datasets in Section 6.4.1. Section 6.4.2 shows the results on synthetic datasets, evaluating clustering accuracy.

6.4.1 Results on real-world datasets

This section evaluates the performance of the proposed method by comparing it with the three implementations. The experiments measured elapsed time from when graph data is ready on the main memory until the clustering result is constructed on the main memory. Note that the data transfer time between the CPU and the GPU was included in the measurement. Real-world datasets were obtained from Stanford Network Analysis Project [74]. Table 6.1 summarizes the five datasets used in the experiments.

Figure 6.6 illustrates the processing time of the four implementations on the five datasets. The figure shows that *GPU-LP* is fastest and successfully accelerates the processing of graph clustering. Concretely, *GPU-LP* outperforms *Louvain* by a factor of approximately 55 on amazon and orkut, and *GPU-LP* is around 70 times faster than *Louvain* on dblp and youtube. In particular, *GPU-LP* achieves a higher speedup of 117 on livejournal.

Compared to *CPU-LP*, *GPU-LP* achieves speedups between 25 and 36. Specifically, 35 times improvements are attained on dblp, youtube, and livejournal. On the other hand, on the largest dataset orkut, the speedup is 25, lower than those on the other datasets. This is because while the computational cost of *CPU-LP* decreases as the iterations proceed, *GPU-LP* retains the computational cost throughout the iterations. Figure 6.7 shows these trends, where the x-axis is the iteration number and the y-axis is the processing time at each iteration in a log scale. The figure indicates that the reduction of processing time is steeper on orkut than on livejournal. As a result, the speedup of *GPU-LP* on orkut become lower than that on livejournal.

The performance of *GPU-LILP* is heavily dependent on datasets. While *GPU-LILP* is slowest on the three datasets, amazon, dblp, and youtube, it is second fastest on livejournal and orkut. This behavior is due to the difference of the skewness in degree distributions. The first three datasets are sparser and more skewed than the other two datasets. Since *GPU-LILP* does not take into account load balancing, it suffers from severe load imbalance. By introduction segmented sort and segmented reduce, *GPU-LP* substantially outperforms *GPU-LILP* by a factor of up to 93.25.

6.4.2 Results on synthetic datasets

This section evaluates the clustering accuracy by using synthetic datasets. Datasets are generated by the *LFR (Lancichinetti–Fortunato–Radicchi)* benchmark [71], which is commonly employed to evaluate graph-clustering algorithms. The LFR benchmark generates graph data with community information. This information is used in our experiments to evaluate accuracy. To measure accuracy, the following three famous metrics are adopted [21]: (1) NMI (normalized mutual information), (2) F-measure, (3) ARI (adjusted Rand index).

NMI measures the similarity between two partitions C and C' on the basis of an

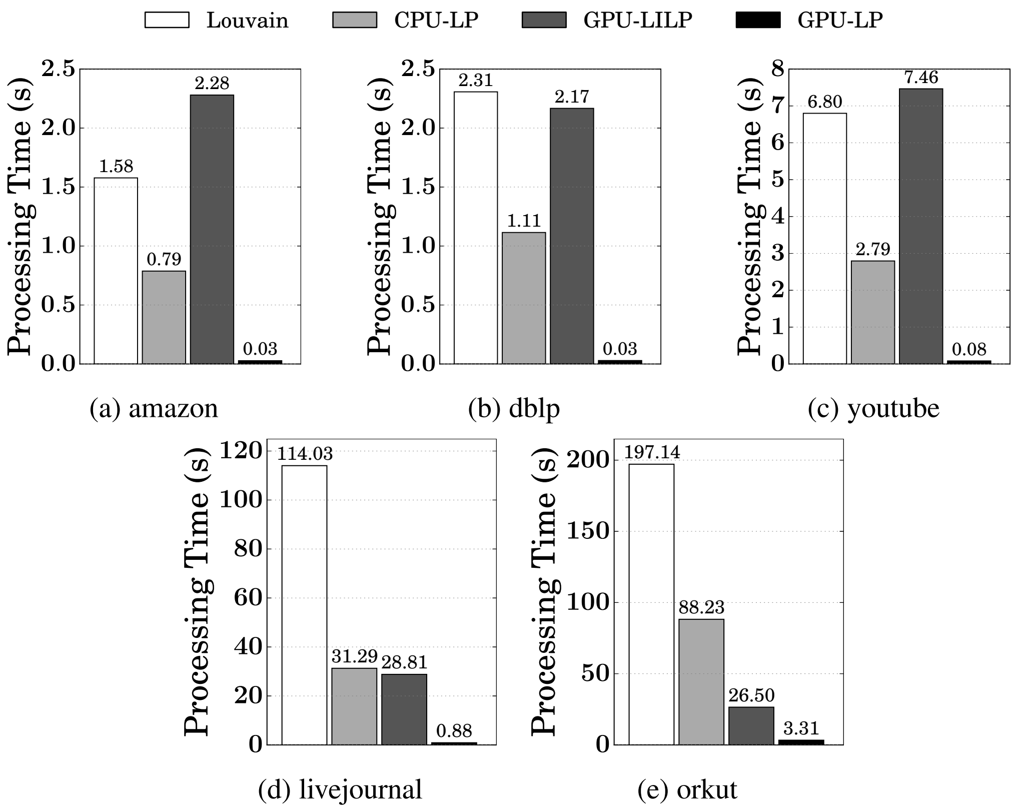


Figure 6.6: Processing times of the four implementations on the five datasets.

information-theoretic approach:

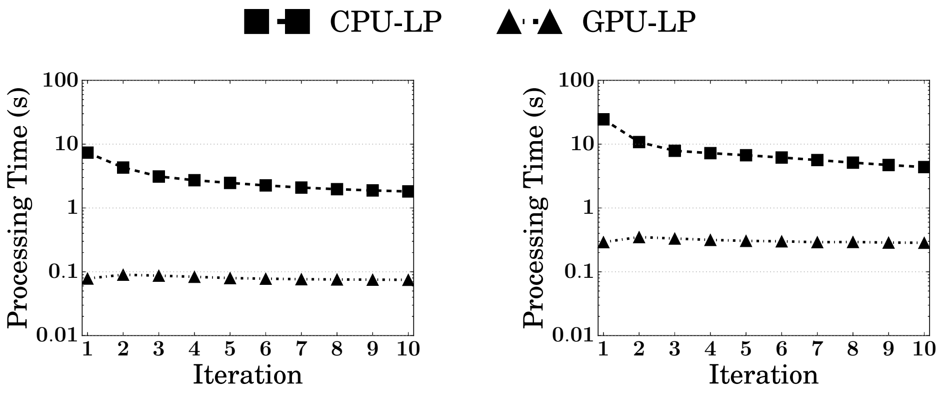
$$\mathrm{NMI}(C, C') = \frac{-2\sum_{i,j} N_{ij}\log(N_{ij}N_t/N_{i\cdot}N_{\cdot j})}{\sum_i N_{i\cdot}\log(N_{i\cdot}/N_t) + \sum_j N_{\cdot j}\log(N_{\cdot j}/N_t)},$$

where N is the confusion matrix whose entry N_{ij} is the number of vertices that appear in both C_i and C_j, $N_{i\cdot}$ and $N_{\cdot j}$ are the sum over the row i and the sum over the column j, respectively, and N_t is the sum over the entire matrix. NMI takes values between 0 and 1, and if two partitions are same, it takes 1.

F-measure measures the similarity of two partitions by finding the largest overlaps between pairs of clusters:

$$\mathrm{F}(C, C') = \frac{1}{n}\sum_{C_i \in C} |C_i| \max_{C'_j \in C'} \frac{2|C_i \cap C'_j|}{|C_i| + |C'_j|}$$

F-measure also takes values between 0 and 1, and F-measures equals 1 in the best case.



(a) livejournal (b) orkut

Figure 6.7: Processing time at each iteration.

ARI is a metric based on pair counting:

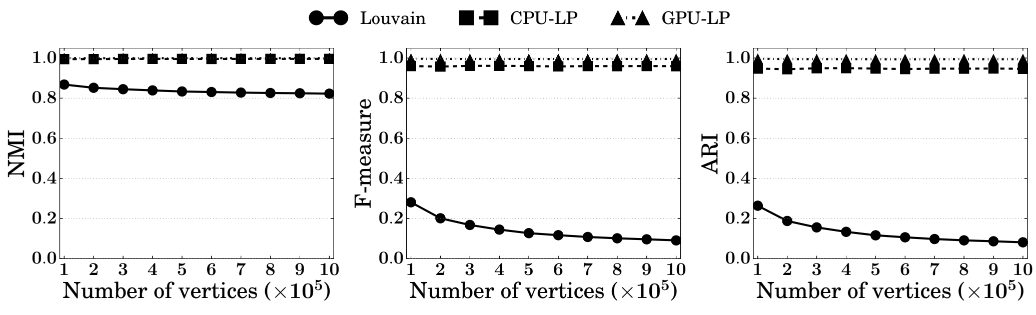
$$\mathrm{ARI}(C, C') = \frac{\sum_{i,j}\binom{N_{ij}}{2} - M}{\frac{1}{2}\left[\sum_i \binom{N_{i\cdot}}{2} + \sum_j \binom{N_{\cdot j}}{2}\right] - M},$$

where N is the confusion matrix and $M = [\sum_i \binom{N_{i\cdot}}{2} \sum_j \binom{N_{\cdot j}}{2}]/\binom{n}{2}$. ARI takes 1 in the base case as with the other metrics.

Figure 6.8 shows the results on synthetic datasets, where the x-axis means the number of vertices and the y-axis shows values of each metric. The data is generated with the average degree of 20 and the maximum degree of 50. Compared to *Louvain*, the implementations of label propagation (i.e., *CPU-LP* and *GPU-LP*) achieve superior accuracy. *CPU-LP* and *GPU-LP* exhibit similar accuracy and are not affected by the number of vertices. On the other hand, the accuracy of *Louvain* deteriorates as the number of vertices increases. This is because the modularity employed in *Louvain* has the limitation, called resolution limit [36] (Section 2.6).

6.5 Summary

This chapter has presented a fast graph clustering algorithm based on parallel label propagation on the GPU. To efficiently process graph clustering, the algorithm of label propagation is firstly transformed into a sequence of data-parallel primitives. In addition, load balancing is taken into account by using the primitives that make the load among threads and blocks well balanced. Experiments on both real-world and



(a) NMI (b) F-measure (c) ARI

Figure 6.8: Results on synthetic datasets.

synthetic datasets show that our proposed method is superior than existing methods, in terms of not only efficiency but also accuracy.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This dissertation has developed GPU-accelerated data analysis techniques, by focusing on effective data structures, data-parallel primitives, and capturing key components. Effective data structures enable achieving high effective memory bandwidth on the GPU; data-parallel primitives help to harness the massive and hierarchical parallelism as well as to balance the workloads among hierarchical threads; and capturing key components enables us to carefully parallelize the bottleneck part of algorithms and dramatically accelerate the processing. On the basis of the techniques, we have made the following contributions:

- This dissertation has introduced GPU-accelerated frequent itemset mining from uncertain data in Chapter 3. Uncertain transaction databases are represented by the ELL format that enables efficient accesses to the data. The pApriori algorithm is parallelized by exploiting data-parallel primitives, namely map, reduce, and filter. The key component of this algorithms is the computation of many repeated convolutions, which are thoroughly accelerated by combining multiple techniques. Consequently, the proposed method outperformed a CPU parallel implementation by a factor of up to 5.5.

- We have presented a novel efficient comparison-sorting method for GPUs in Chapter 4. The method devise the data layout that takes into account memory alignment, thereby enabling more coalesced accesses, and it also relies on several data-parallel primitives such as scan and scatter. The key component is to reduce the memory access cost, which is achieved by combining two existing algorithms, samplesort and merge sort. As a result, the proposed method improved throughput by 32% over an existing GPU-based method.

- Canopy clustering on the GPU has been proposed in Chapter 5. The proposed method heavily depends on the CSR format, to represent a set of canopies and grid index. Data-parallel primitives, including map, reduce, filter, and gather, are exploited throughout the algorithms. For further acceleration over simple parallelization, the method constructs grid index on the GPU, thereby reducing the number of distance computations and providing high parallelism to the GPU. Experiments showed that the GPU implementation was at most 2.5 times faster than CPU parallel implementation, even if the CPU counterparts used two octa-core processors.
- GPU-accelerated label propagation for graph clustering has been described in Chapter 6. The proposed method utilizes the CSR format to represent graph data, because of low space cost and suitability for data-parallel primitives. The algorithm of label propagation is transformed into a sequence of data-parallel primitives such as map, reduce, scan, and gather. The key component here is load balancing, which is achieved by employing the primitives that make the load among threads and blocks well balanced. GPU-accelerated label propagation achieved 30 times higher performance on average than a CPU implementation of label propagation.

Observing these results, we believe that our GPU-acceleration techniques help to improve the performance of not only the specific four problems but also a wider range of applications.

7.2 Future work

Several future directions are possible: (1) handling larger data than the GPU memory, (2) using multiple GPUs and GPU clusters, and (3) exploiting heterogeneous resources such as CPUs and GPUs. The first direction requires to devise methods that partition data on the main memory, transfer a part of data to the GPU, process it, and return the result to the main memory. These methods are particularly important among the future directions, because if the methods are achieved, they can be relatively easily extended to use multiple GPUs, GPU clusters, and heterogeneous resources. Thus the following provides descriptions of the first directions for the four problems.

Frequent itemset mining: In our method, inclists—each of which is an integer array storing whether a candidate is included in transactions or not—are the largest data maintained on the GPUs, and they are associated with candidates. Thus, if candidates are partitioned by some means, larger-scale data can be handled. Actually, we have proposed methods based on this idea using multiple GPUs and GPU clusters [70].

Sorting: Our sorting method firstly partitions data into *k* buckets. If we can transfer and sort each bucket separately, we achieve sorting of larger data than the GPU memory. However, the partitioning is done on the GPU under the assumption that the data fit into the GPU. Thus we need either an efficient method of partitioning on the main memory or a method that uses GPUs by intelligently overlapping partitioning and transfer.

Canopy clustering: Since our method uses grid index, the data can be easily partitioned on the basis of grid cells if grid index is constructed. In the case that the grid index cannot be constructed on the GPU because of running out of GPU memory, the grid index can be constructed on the main memory, because the construction does not take so long time even if CPUs are used.

Graph clustering: Adapting our graph clustering method to larger data is relatively easy. One way is to partition a set of vertices into multiple sets and process each set separately. In our preliminary evaluation, this method with partitioning achieved comparable performance to the method without partitioning.

As more general and long-term future work, development of a programming model can be considered. When utilizing GPU clusters, we need to learn several parallel programming models such as CUDA, OpenMP, and MPI, and this is greatly burdensome, costly, and difficult. To ease the development, a novel programming model should be devised, e.g., on the basis of data-parallel primitives exploited in this dissertation.

Bibliography

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, “A Tree Projection Algorithm for Generation of Frequent Item Sets,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 61, no. 3, pp. 350–371, March 2001.

[2] C. C. Aggarwal and P. S. Yu, “A Survey of Uncertain Data Algorithms and Applications,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 21, no. 5, pp. 609–623, May 2009.

[3] R. Agrawal, T. Imieliński, and A. Swami, “Mining Association Rules between Sets of Items in Large Databases,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 207–216, 1993.

[4] R. Agrawal and J. C. Shafer, “Parallel Mining of Association Rules,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 8, no. 6, pp. 962–969, December 1996.

[5] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” in *Proceedings of the 20th Interantional Conference on Very Large Data Bases (VLDB)*, pp. 487–499, 1994.

[6] R. R. Amossen and R. Pagh, “A New Data Layout for Set Intersection on GPUs,” in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 698–708, 2011.

[7] A.-L. Barabási and R. Albert, “Emergence of Scaling in Random Networks,” *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[8] S. Baxter, “Modern GPU,” `http://nvlabs.github.io/moderngpu/`, 2013.

[9] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-first Search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 12:1–12:10, 2012.

[10] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pp. 18:1–18:11, 2009.

[11] T. Bernecker, H.-P. Kriegel, M. Renz, F. Verhein, and A. Zuefle, "Probabilistic Frequent Itemset Mining in Uncertain Databases," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 119–128, 2009.

[12] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research (JMLR)*, vol. 3, pp. 993–1022, March 2003.

[13] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," in *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 3–16, 1991.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, October 2008.

[15] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based Clustering using Graphics Processors," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pp. 661–670, 2009.

[16] A. Buluç, J. R. Gilbert, and C. Budak, "Solving Path Problems on the GPU," *Parallel Computing*, vol. 36, no. 5–6, pp. 241–253, June 2010.

[17] T. Calders, C. Garboni, and B. Goethals, "Approximation of Frequentness Probability of Itemsets in Uncertain Data," in *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM)*, pp. 749–754, 2010.

[18] T. Calders, C. Garboni, and B. Goethals, "Efficient Pattern Mining of Uncertain Data with Sampling," in *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pp. 480–487, 2010.

[19] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast Support Vector Machine Training and Classification on Graphics Processors," in *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pp. 104–111, 2008.

[20] P. Cazzaniga, F. Ferrara, M. S. Nobile, D. Besozzi, and G. Mauri, "Parallelizing Biochemical Stochastic Simulations: A Comparison of GPUs and Intel Xeon Phi Processors," in *Proceedings of the 13th International Conference on Parallel Computing Technologies (PaCT)*, pp. 363–374, 2015.

[21] M. Chen, K. Kuzmin, and B. K. Szymanski, "Community Detection via Maximization of Modularity and Its Variants," *IEEE Transactions on Computational Social Systems (TCSS)*, vol. 1, no. 1, pp. 46–65, March 2014.

[22] C.-K. Chui, B. Kao, and E. Hung, "Mining Frequent Itemsets from Uncertain Data," in *Proceedings of the 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pp. 47–58, 2007.

[23] G. Cordasco and L. Gargano, "Label Propagation Algorithm: A Semi-synchronous Approach," *International Journal of Social Network Mining*, vol. 1, no. 1, pp. 3–26, 2012.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[25] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored Approach for Training Kernelized SVMs," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 805–813, 2011.

[26] M. Dash, S. Petrutiu, and P. Scheuermann, "pPOP: Fast yet accurate parallel hierarchical clustering using partitioning," *Data & Knowledge Engineering*, vol. 61, no. 3, pp. 563–578, June 2007.

[27] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 349–359, 2014.

[28] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI)*, pp. 137–150, 2004.

[29] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 7, pp. 940–952, July 2013.

[30] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier, "All-Pairs Shortest Path Algorithms for Planar Graph for GPU-accelerated Clusters," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 85, pp. 91–103, November 2015.

[31] H. N. Djidjev and M. Onus, "Scalable and Accurate Graph Clustering and Community Structure Detection," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 5, pp. 1022–1029, May 2013.

[32] E. Duriakova, N. Hurley, D. Ajwani, and A. Sala, "Analysis of the Semi-synchronous Approach to Large-scale Parallel Community Finding," in *Proceedings of the Second ACM Conference on Online Social Networks (COSN)*, pp. 51–62, 2014.

[33] Z.-G. Fan, Y. Wu, and B. Wu, "Maximum Normalized Spacing for Efficient Visual Clustering," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 409–418, 2010.

[34] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent Itemset Mining on Graphics Processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN)*, pp. 34–42, 2009.

[35] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3–5, pp. 75–174, February 2010.

[36] S. Fortunato and M. Barthlemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 104, no. 1, pp. 36–41, December 2007.

[37] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, March 1977.

[38] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *Proceedings of Workshop on Graph Data Management Experiences and Systems (GRADES)*, pp. 2:1–2:6, 2014.

[39] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, Second Edition, Prentice Hall Press, 2008.

[40] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks,” in *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA)*, pp. 319–333, 2008.

[41] M. Goldfarb, Y. Jo, and M. Kulkarni, “General Transformations for GPU Execution of Tree Traversals,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 10:1–10:12, 2013.

[42] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 17–30, 2012.

[43] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, “GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management,” in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 325–336, 2006.

[44] O. Green, R. McColl, and D. A. Bader, “GPU Merge Path: A GPU Merging Algorithm,” in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, pp. 331–340, 2012.

[45] T. L. Griffiths and M. Steyvers, “Finding scientific topics,” *Proceedings of the National Academy of Sciences (PNAS)*, vol. 101, no. suppl 1, pp. 5228–5235, April 2004.

[46] S.-W. Ha and T.-D. Han, “A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 12, pp. 2324–2333, December 2013.

[47] J. Han, J. Pei, and Y. Yin, “Mining Frequent Patterns without Candidate Generation,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1–12, 2000.

[48] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, Third Edition, Morgan Kaufmann Publishers, 2011.

[49] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *Proceedings of the 14th International Conference on High Performance Computing (HiPC)*, pp. 197–208, 2007.

[50] M. Harris, "Optimizing Parallel Reduction in CUDA," http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/reduction/doc/reduction.pdf.

[51] M. Harris, "Parallel Prefix Sum (Scan) with CUDA," http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/scan/doc/scan.pdf, 2009.

[52] J. He, M. Lu, and B. He, "Revisiting Co-Processing for Hash Joins on the Coupled CPU–GPU Architecture," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 10, pp. 889–900, August 2013.

[53] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, pp. 46:1–46:12, 2007.

[54] B. He, K. Yang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational Joins on Graphics Processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 511–524, 2008.

[55] G. He, C. Li, H. Chen, X. Du, and H. Feng, "Using Graphics Processors for High Performance SimRank Computation," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 9, pp. 1711–1725, September 2012.

[56] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational Query Coprocessing on Graphics Processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, pp. 21:1–21:39, December 2009.

[57] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-Oblivious Parallelism for In-Memory Column-Stores," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 9, pp. 709–720, July 2013.

[58] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, no. 5786, pp. 504–507, July 2006.

[59] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 78–88, 2011.

[60] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA Graph Algorithms at Maximum Warp,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 267–276, 2011.

[61] S. S. Huang, T. J. Green, and B. T. Loo, “Datalog and Emerging Applications: An Interactive Tutorial,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)* pp. 1213–1216, 2011.

[62] G. Jeh and J. Widom, “SimRank: A Measure of Structural-context Similarity,” in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 538–543, 2002.

[63] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, “GPU Join Processing Revisited,” in *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN)*, pp. 55–62, 2012.

[64] G. J. Katz and J. T. Kider, Jr, “All-Pairs Shortest-Paths for Large Graphs on the GPU,” in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH)*, pp. 47–55, 2008.

[65] J. Kim, S.-G. Kim, and B. Nam, “Parallel multi-dimensional range query processing with R-trees on GPU,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 8, pp. 1195–1207, August 2013.

[66] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, “Designing Fast Architecture-sensitive Tree Search on Modern Multicore/Many-core Processors,” *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 4, pp. 22:1–22:34, December 2011.

[67] J. M. Kleinberg, “Authoritative Sources in a Hyperlinked Environment,” *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, September 1999.

[68] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison Wesley, Second Edition, 1998.

[69] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, “K-Means for Parallel Architectures Using All-Prefix-Sum Sorting and Updating Steps,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 8, pp. 1602–1612, August 2013.

[70] Y. Kozawa, T. Amagasa, and H. Kitagawa, “Probabilistic Frequent Itemset Mining on a GPU Cluster,” *IEICE Transactions on Information and Systems*, vol. E97-D, no. 4, pp. 779–789, April 2014.

[71] A. Lancichinetti, S. Fortunato, and F. Radicchi, “Benchmark graphs for testing community detection algorithms,” *Physical Review E*, vol. 78, no. 4, p. 046110, October 2008.

[72] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, May 2015.

[73] N. Leischner, V. Osipov, and P. Sanders, “GPU Sample Sort,” in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–10, 2010.

[74] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” `http://snap.stanford.edu/data`, June 2014.

[75] C. K.-S. Leung and S. K. Tanbeer, “Fast Tree-Based Mining of Frequent Itemsets from Uncertain Data,” in *Proceedings of the 17th International conference on Database Systems for Advanced Applications (DASFAA)*, pp. 272–287, 2012.

[76] C. K.-S. Leung and Y. Hayduk, “Mining Frequent Patterns from Uncertain Data with MapReduce for Big Data Analytics,” in *Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 440–455, 2013.

[77] C. K.-S. Leung, M. A. F. Mateo, and D. A. Brajczuk, “A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data,” in *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pp. 653–661, 2008.

[78] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft, “Towards real-time community detection in large networks,” *Physical Review E*, vol. 79, no. 6, p. 066107, June 2009.

[79] Q. Li, P. Wang, W. Wang, H. Hu, Z. Li, and J. Li, “An Efficient K-means Clustering Algorithm on MapReduce,” in *Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 357–371, 2014.

[80] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-Means algorithm by GPUs," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, March 2013.

[81] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "Parallel multitask cross validation for Support Vector Machine using GPU," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 3, pp. 293–302, March 2013.

[82] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-Growth for Query Recommendation," in *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys)*, pp. 107–114, 2008.

[83] H. Liu and H. H. Huang, "Enterprise: Breadth-first Graph Traversal on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 68:1–68:12, 2015.

[84] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of Frequent Itemset Mining on Multiple-Core Processor," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pp. 1275–1285, 2007.

[85] C. Lomont, "Introduction to Intel® Advanced Vector Extensions," `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`, 2011.

[86] L. Luo, M. Wong, and W.-m. Hwu, "An Effective GPU Implementation of Breadth-first Search," in *Proceedings of the 47th Design Automation Conference (DAC)*, pp. 52–55, 2010.

[87] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA Solutions for the SSSP Problem," in *Proceedings of the 9th International Conference on Computational Science (ICCS)*, pp. 904–913, 2009.

[88] E. Mastrostefano and M. Bernaschi, "Efficient breadth first search on multi-GPU systems," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 9, pp. 1292–1305, September 2013.

[89] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked United Algorithm for the All-Pairs Shortest Paths Problem on Hybrid CPU-GPU Systems," *IEICE Transactions on Information and Systems*, vol. 95-D, no. 12, pp. 2759–2768, 2012.

[90] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient Clustering of High-dimensional Data Sets with Application to Reference Matching," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 169–178, 2000.

[91] A. McLaughlin and D. A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 572–583, 2014.

[92] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing," *Parallel Processing Letters*, vol. 21, no. 2, pp. 245–272, June 2011.

[93] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 117–128, 2012.

[94] D. Merrill, "CUB," `http://nvlabs.github.io/cub/`, 2015.

[95] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 69:1–69:35, July 2015.

[96] R. Nasre, M. Burtscher, and K. Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 463–474, 2013.

[97] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, February 2004.

[98] NVIDIA, "CUDA C Programming Guide." `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2015

[99] NVIDIA, "CUFFT Library User Guide," `http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf`, 2012.

[100] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge Path - Parallel Merging Made Simple," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 1611–1618, 2012.

[101] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all-pairs shortest paths using the GPU," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, April 2012.

[102] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, pp. 607–609, June 1996.

[103] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and Resource-Aware Mining of Frequent Sets," in *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*, pp. 338–345, 2002.

[104] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[105] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, Stanford InfoLab, November 1999.

[106] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos, "Community Detection in Social Media," *Data Mining and Knowledge Discovery*, vol. 24, no. 3, pp. 515–554, May 2012.

[107] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li, "Parallel Data Mining for Association Rules on Shared-Memory Systems," *Knowledge and Information Systems*, vol. 3, no. 1, pp. 1–29, February 2001.

[108] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 62:1–62:11, 2012.

[109] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "A novel sorting algorithm for many-core architectures based on adaptive bitonic sort," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 227–237, 2012.

[110] J. C. Platt, "Fast Training of Support Vector Machines Using Sequential Minimal Optimization," *Advances in Kernel Methods*, pp. 185–208, The MIT Press, 1999.

[111] X. Que, F. Checconi, F. Petrini, and J. Gunnels, "Scalable Community Detection with the Louvain Algorithm," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 28–37, 2015.

[112] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, September 2007.

[113] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale Deep Unsupervised Learning using Graphics Processors," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pp. 873–880, 2009.

[114] J. R. Rice, "ELLPACK 77 Users' Guide," *Computer Science Technical Reports. Paper 219.*, `http://docs.lib.purdue.edu/cstech/219`.

[115] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–10, 2009.

[116] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 351–362, 2010.

[117] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner, "Scalable Frequent Itemset Mining on Many-core Processors," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN)*, pp. 3:1–3:8, 2013.

[118] S. A. A. Shalom and M. Dash, "Efficient Partitioning Based Hierarchical Agglomerative Clustering Using Graphics Accelerators with CUDA," *International Journal of Artificial Intelligence & Applications*, vol. 4, no. 2, pp. 13–33, March 2013.

[119] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "Fast Algorithm for Modularity-Based Graph Clustering," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI)* pp. 1170–1176, 2013.

[120] C. Silvestri and S. Orlando, "gpuDCI: Exploiting GPUs in Frequent Itemset Mining," in *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 416–425, 2012.

[121] J. Soman and A. Narang, "Fast Community Detection Algorithm with GPUs and Multicore Architectures," in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 568–579, 2011.

[122] E. Soroush, M. Balazinska, and D. Wang, "ArrayStore: A Storage Manager for Complex Parallel Array Processing," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 253–264, 2011.

[123] T. R. Stovall, S. Kockara, and R. Avci, "GPUSCAN: GPU-Based Parallel Structural Clustering Algorithm for Networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 12, pp. 3381–3393, December 2015.

[124] L. Sun, R. Cheng, D. W. Cheung, and J. Cheng, "Mining Uncertain Data with Probabilistic Guarantees," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 273–282, 2010.

[125] Y. W. Teh, D. Newman, and M. Welling, "A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation," in *Advances in Neural Information Processing Systems 19 (NIPS)*, pp. 1353–1360, 2007.

[126] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative Performance Analysis of Intel® Xeon Phi™, GPU, and CPU: A Case Study from Microscopy Image Analysis," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1063–1072, 2014.

[127] G. Teodoro, N. Mariano, W. Meira Jr., and R. Ferreira, "Tree Projection-based Frequent Itemset Mining on Multicore CPUs and GPUs," in *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 47–54, 2010.

[128] Y. Tong, L. Chen, Y. Cheng, and P. S. Yu, "Mining Frequent Itemsets over Uncertain Databases," *Proceddings of the VLDB Endowment (PVLDB)*, vol. 5, no. 11, pp. 1650–1661, July 2012.

[129] T. T. Tran, Y. Liu, and B. Schmidt, "Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi," *Parallel Computing*, vol. 54, pp. 128–138, May 2016.

[130] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU," in *Proceedings of the 21th ACM symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 11:1–11:12, 2016.

[131] M. Wang, C. Wang, J. X. Yu, and J. Zhang, "Community Detection in Social Networks: An In-depth Benchmarking Study with a Procedure-oriented Framework," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 10, pp. 998–1009, June 2015.

[132] L. Wang, D. W.-L. Cheung, R. Cheng, S. D. Lee, and X. S. Yang, "Efficient Mining of Frequent Item Sets on Large Uncertain Databasesn," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 12, pp. 2170–2183, December 2012.

[133] M. Wasif and P. Narayanan, "Scalable Clustering using Multiple GPUs," in *Proceedings of the 2011 18th International Conference on High Performance Computing (HiPC)*, pp. 1–10, 2011.

[134] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme Scale Density-based Clustering Using a Tree-based Network of GPGPU Nodes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 84:1–84:11, 2013.

[135] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red Fox: An Execution Environment for Relational Query Processing on GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 44:44–44:54, 2014.

[136] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 107–118, 2012.

[137] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger, "SCAN: A Structural Clustering Algorithm for Networks," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 824–833, 2007.

[138] F. Yan, N. Xu, and Y. Alan Qi, "Parallel Inference for Latent Dirichlet Allocation on Graphics Processing Units," in *Advances in Neural Information Processing Systems 22 (NIPS)*, pp. 2134–2142, 2009.

[139] S. Yan, G. Long, and Y. Zhang, “StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 229–238, 2013.

[140] X. Yang, S. Parthasarathy, and P. Sadayappan, “Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 4, pp. 231–242, January 2011.

[141] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, “High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs,” in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–10, 2010.

[142] J. Zhong and B. He, “Medusa: Simplified Graph Processing on GPUs,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 6, pp. 1543–1552, June 2014.

List of Publications

Publications related to the dissertation

Refereed journal papers

- Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa "An Efficient Sorting Algorithm on GPUs via Data Partitioning and Cooperative Merge," *DBSJ Japanese Journal*, vol. 13-J, no. 2, pp. 1–6, February 2015. (in Japanese with English abstract)

- Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Probabilistic Frequent Itemset Mining on a GPU Cluster," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 4, pp. 779–789, April 2014.

Refereed international conference papers

- Yusuke Kozawa, Fumitaka Hayashi, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Parallel Canopy Clustering on GPUs," in *Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 334–348, Valencia, Spain, September 1–4, 2015.

- Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Parallel and Distributed Mining of Probabilistic Frequent Itemsets Using Multiple GPUs," in *Proceedings of the 24th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 145–152, Prague, Czech Republic, August 26–30, 2013.

- Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "GPU Acceleration of Probabilistic Frequent Itemset Mining from Uncertain Databases," in *Proceedings of the 21st ACM International Conference on Information and*

Knowledge Management (CIKM), pp. 892–901, Maui, Hawaii, October 29–November 2, 2012.

- Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "Fast Frequent Itemset Mining from Uncertain Databases Using GPGPU," in *Proceedings of the Fifth International VLDB Workshop on Management of Uncertain Data (MUD)*, pp. 17–24, Seattle, Washington, August 29, 2011.

Other publications

Refereed journal papers

- Fumitaka Hayashi, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "GPU Acceleration of Canopy Clustering," *DBSJ Japanese Journal*, vol. 13-J, no. 2, pp. 13–18, February 2015. (in Japanese with English abstract)

Refereed international conference papers

- Mateus S. H. Cruz, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "GPU Acceleration of Set Similarity Join," in *Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 384–398, Valencia, Spain, September 1–4, 2015.

- Jun Hwang, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, "GPU Acceleration of Similarity Search for Uncertain Time Series," in *Proceedings of the 17th International Conference on Network-Based Information Systems (NBiS)*, pp. 627–632, Salerno, Italy, September 10–12, 2014.