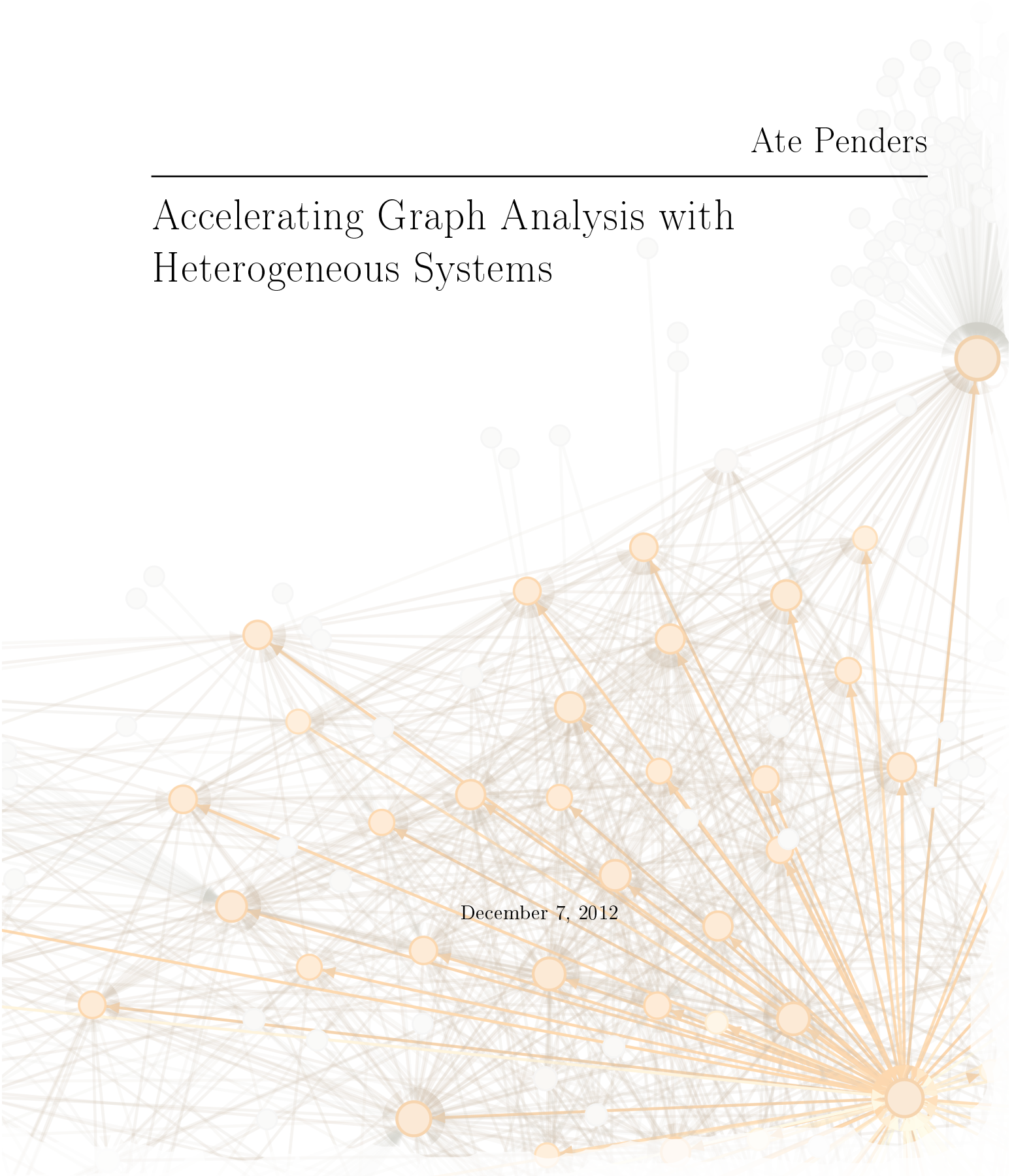
Ate Penders

Accelerating Graph Analysis with Heterogeneous Systems



December 7, 2012

Ate Penders

Accelerating Graph Analysis with Heterogeneous Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING
by

Ate Penders

December 7, 2012

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Software and Computer Technology
Parallel and Distributed Systems Group

Abstract

Data analysis is a rising field of interest for computer science research due to the growing amount of information that is digitally available. This increase in data has as direct consequence that any analysis is significantly complex. By using structured representations for the data sets, like graphs, the analysis becomes feasible, but is still time-consuming. In this project, the focus is on the reduction of the computational time for data analysis, with the introduction of accelerators. Accelerators are specialized hardware components that assist the general processing unit in performing (parts of) the task at hand. In particular, we focus on the use of General Purpose Graphical Processing Units (GPGPU) to help speedup the analysis. GPUs are specifically designed for representing and manipulating graphical data which invoke the processing of large chunks of data, GPUs are designed with large numbers of concurrent processing units and thus have a high potential of improving performance.

In this project, we show the impact of using GPGPUs for both simple and more complex analysis, varying from small to large data sets, with the use of a programming model called OpenCL. We compare the performance of using accelerators against the traditional CPU-based implementation. Due to the inter-platform portability of the OpenCL model, such comparison can be performed without having to alter the algorithm.

The use of accelerators is expected to become beneficial for analysis that require large computational power. For example, search algorithms (that require little to no computation) are not expected to profit from accelerators, while the more complex, centrality analysis is expected to have significantly more benefit from accelerators. Our results clearly shows this shift of performance improvement when algorithms further utilize the potential of accelerators, because the analysis grows in size and/ or complexity. We conclude that the use of accelerators in graph processing is promising, despite the large variation in performance improvement and its strong dependency on data, algorithm and hardware.

Thesis Committee:

Supervisor:	Dr. Ir. Ana Lucia Varbanescu, Faculty EEMCS, PDS, TU Delft
Committee Member:	Prof. Dr. Ir. Henk Sips, Faculty EEMCS, PDS, TU Delft
Committee Member:	Dr. Ir. Alexandru Iosup, Faculty EEMCS, PDS, TU Delft
Committee Member:	Dr. Ir. Stefan Dulman, Faculty EEMCS, ES, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Many real systems and structures in our lives can be regarded as complex networks. For example, decision making can be mapped onto a network of choices, where every choice leads to a set of follow-up choices, eventually leading to a decision. Our social environments can also be mapped on a network of connections of friends, friends of friends and so on. With the digitalization of many systems (like social life is digitalized in social media), the field of data analysis experienced a significant growth in interest. Because of the large size and data dependencies, a substantial level of complexity is invoked in the analysis, making structured data representations a nontrivial step towards feasible data analysis. Graphs are an example of a structured data representation that is often used for its flexibility and abstraction over complexity. A graph representation consists of a set of actors in the network, and a set of relations between these actors. For example, in social networks the set of actors is the population and the relations can vary from friendship, work-relation, to a common interest or shared place of residence. Such a data structure eases the complexity of an analysis, still the increasing size of the available data collections require extensive computations times for any type of analysis. For example, finding the most important person in a social network has a $\omega(n^3)$ bottleneck (where n is the number of persons in the network), when using a traditional single-core processor. [14] Increasing the size of the network will increase the response time significantly. Thus, sequential approaches to data analysis are not feasible performance wise.

The introduction of parallel systems helped in reducing the time-complexity of this type of data analysis significantly. Instead of having a single large and complex processor to perform the task, parallel systems use multiple smaller processors, that work together in solving the task at hand. In this way, performance improves and power consumption is more controllable (processors that are no longer needed can be placed in stand-by to save power). This increased performance comes at the cost of complexity and scalability: the data and the operations often have dependencies that require communication between the processors to exchange information. By introduction of redundancy (i.e. additional

operations), the dependencies can be reduced, consequently increasing the complexity. In other words, parallel systems will improve performance of an algorithm, but this improvement is typically not linear with the increase in the number of processors (unless the data and the operations are both completely independent).

In the field of parallel systems, a growing trend is using accelerators – specialized hardware components that assist the general processing unit in performing (parts of) the task at hand – for general purpose scientific computing. An accelerator is a device with a high level of parallelism: it, typically, has a large number of simple processors that can be scheduled to work together on a single task. A frequently used accelerator is a Graphical Processing Unit (GPU). A GPU has a large amount of processing units, that are designed to work concurrently on the processing of large amounts of data. GPUs are, traditionally, designed solely for representing and manipulating graphical data: many graphical computations consists of a large number of simple operations, not requiring sophisticated processing units to perform the task at hand. Recent GPU designs show a large improvement in the capacity of the processing units, making them suitable for performing scientific programming as well. Due to the fine-grained level of parallelism in GPUs, they show great potential in improving performance of scientific computing workloads.

This trend of using accelerators also starts to appear in performing graph analysis (i.e. data analysis, with the data represented as graphs). Because graphs – data sets that can be represented as relationship networks – tend to be very irregular in structure, the performance differs from that of scientific computing applications, which are generally regular and computation-driven. Therefore, a parallel system that works well for general purpose scientific computing, does not necessarily work well for graph analysis. In this project we focus on the use of GPUs to accelerate graph analysis, and prove by example the positive impact these accelerators have on graph analysis, when compared with sequential and traditional parallel systems.

1.1 Motivation

With the rapidly increasing availability and size of knowledge bases (an information repository for data and relations of that data to other data), we expect the need for graph analysis to show a growth at least linear to this increase. In the same time, the graph representing the knowledge base explodes in time and complexity, making each graph analysis very time consuming. This further motivates the need for more parallel computing. As accelerators become vitally important in the field of parallel computing, either as stand-alone solution or as part of a heterogeneous cluster of parallel systems, it is essential to understand how these two opportunities – large scale complex graph analysis and increasingly powerful accelerators – can be made to successfully work together.

From the range of possible accelerators, we choose to use (general purpose) GPUs, for their fine-grained parallel structure and relatively high performance of the individual processors. Furthermore, GPUs are commercially available, and are easily integrated in an existing system. For the implementation of accelerated graph analysis, we use a general programming model OpenCL. This model allows the use of various systems, without the need to alter the design. Thus, OpenCL allows us to have a comparison between different parallel systems (multi-CPU and accelerated). This thesis only limits to GPU accelerators, since we believe that many other accelerators require additional research before they can be used for general purpose scientific computing. For example, Altera one of the big vendors of FPGAs, only recently (August 2012) published a newsletter claiming to successfully map programs from the OpenCL language onto its FPGAs, allowing these FPGAs to be used as accelerators with OpenCL [16].

1.2 Research Questions

With large-scale graphs (i.e. large data sets), any analysis will have the problem of consuming large amounts of time to finish. Parallel computing helps in reducing this execution time, making the graph analysis more feasible. In parallel computing, the traditional approach is to use an arbitrary number of complex general purpose CPUs and send each of these a big chunk of the problem. Recent trends show a more heterogeneous approach for parallel systems, where various types of processors collaborate on the task at hand. The problem with using heterogeneous systems, is exactly the part where the processors differ from the 'traditional' CPUs (i.e. they differ in computational complexity and power, making them perform at different speeds). In this thesis, we study the behavior of heterogeneous systems that use graphical cards to accelerate parallel computing for graph analysis and try to get a better insight on the impacts these accelerators have on performance.

By intuition, the introduction of more parallelism is better for performance (i.e. the sequences of work are reduced, which should lead to less time to execute). However, the improvements rarely follow a linear behavior due to dependencies of both data and computations, which are application and/ or algorithm dependent. Not just the dependencies prevent more parallelism to be effective, but also the architecture of the parallel system has a major impact. For example, shared memory can help improving performance, if the data in the memory can be used by all the processes, but can also decrease performance, if all the processes want different data to be placed in the shared memory. In this thesis, we address the problem of data dependency and data accesses from the perspective of the influence of graph representation on performance. In other words: how can we change the graph representation such that the performance improves?
Furthermore, since the data dependencies and access patterns influence the performance,

the data itself has a major impact on the performance. Each graph (i.e., data set) has certain properties with respect to the structure of the relations between nodes (e.g., the number of relations each nodes has, or the average number of relations). Therefore, matching graph properties with the performance of the graph analysis on a specific parallel system is highly desirable. But can we really predict the performance of graph analysis based on graph density and structure?

To summarize, this thesis focuses on the following research questions:

- What is the impact on performance when accelerating graph analysis using graphical processing units (GPUs)?
- Do graph representations impact the performance of graph analysis?
- Can we predict the performance gain of graph analysis based on graph density and structure?

1.3 Approach

This thesis focuses entirely on the performance of (accelerated) graph analysis. We use three levels of complexity to examine the performance of three different parallel systems: one multi-core CPU and two GPU accelerated systems. Our three applications represent three different levels of complexity. We start with a fairly simple graph analysis that constructs a search tree from the graph. Next, we increase the complexity by conducting a large number of searches on the same graph, each resulting in an individual search tree. Thus, we reuse our first algorithm, and increase the workload complexity. Finally, we introduce additional reasoning to each individual search: the search tree is used to retrieve statistical information on the centrality of nodes.

Our approach is intended to understand how the different complexity levels impact performance and, more specific, how are the GPU accelerated platforms reacting on these changes. Therefore, our studies do not focus on peak performance (i.e. we do not search for the absolute best achievable execution time). Instead, by reusing the algorithms in more complex settings, we look for patterns in the performance. We use a range of a few thousand to a few million nodes (actors or information nodes in the network) to provide a better overview of the behavior of the various systems. With the use of a programming model called OpenCL, we compare the performance of multi-core CPUs against the use of accelerators. Due to the inter-platform portability of the OpenCL model we can do this without altering the algorithm.

For the implementation, we are not looking for the maximum achievable performance, but aim at a fair comparison between the various systems (accelerated and

non-accelerated). Note that the different systems have completely different architectures, making it impossible to have a completely fair comparison (i.e. each architecture will most likely react slightly different on a specific sequence of operations). Although OpenCL provides numerous of optimization techniques for specific systems, we choose to use only those techniques, which will lead to a fair comparison.

To summarize: in this thesis, attempts are made to compare three different parallel systems on their performance on graph analysis. Both algorithm complexity and data set complexity (i.e., graph size, density, and other properties) are taken into account. Effort is made to keep the implementations on the different machines similar, at the cost of achieving peak performance.

1.4 Thesis Structure

As this thesis intends to study the suitability of graph analysis on accelerated systems and to compare their performance against traditional parallel systems, knowledge of graph analysis and the implementation architecture is required. Chapter 2 outlines a brief introduction to graph analysis, presents a preliminary reasoning on the expected behavior, and describes the important aspects of the OpenCL model for programming heterogeneous systems. In Chapter 3, prior research is listed on the topics related to our study. In the list, we briefly address representing work and how it differs from or influences our work. In Chapter 4, a first graph analysis application is presented, with various experiments we conducted to evaluate our design choices and their impacts on performance. A more complex and time consuming graph analysis is described in Chapter 5. This chapter addresses the power of accelerators and its limitations by using examples of designs that are not feasible because of insufficient resources. These boundaries are analyzed and visualized in the experimental section of the chapter. Chapter 6 presents a more exhaustive graph analysis, on which a similar boundary analysis is applied. Each chapter contains a small discussion on the measured differences in performance of the systems. These discussions are further extended in Chapter 7, where we analyze in more depth the behavior of the different systems as the complexity of the algorithms increases. Chapter 8 provides our conclusions, the main limitations we found in this study, and promising directions for future research.

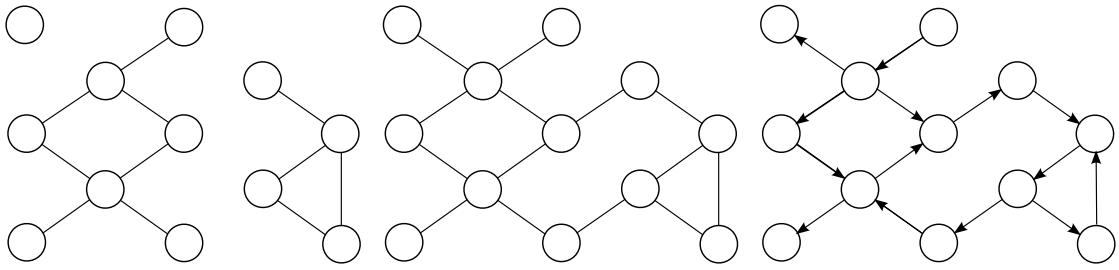
Chapter 2

Background

2.1 Large-scale Graphs

For interpreting and processing large amounts of data, a clear representation is desired. Graphs are such structures, widely used for representing interrelated data in communication networks, computational biology, circuit modeling, social networks and transportation networks. A graph consists of collections of two distinctive elements: (i) vertices, holders of the actual information in the system, and (ii) edges, relationship indicators between pairs of vertices. The systematic approach of distinguishing information and relation allows graphs to be very flexible to changes in the data.

Since a graph has no predefined structure, in the sense that it consists of an arbitrary number of vertices that are connected using an arbitrary number of edges, graphs are categorized based on the connectivity of the different vertices. The specification of a graph does no have any notion on whether each vertex should have an (in)direct connection to all other vertices, hence a first distinction is between *connected* and *unconnected* graphs. An *unconnected* graph is a collection of various *connected* graphs, where *connected* refers to each vertex having an (in)direct connection to all other vertices (See



(a) Unconnected graph (b) Connected undirected graph (c) Connected directed graph

Figure 2.1: Common characterizations of graphs.

Figure 2.1a and 2.1b, respectively). Another common categorization is the direction of edges: if a relation exists between A and B, does this imply that there is the same relation between B and A? For example, in a road network with an edge $A \rightarrow B$, cars can travel from A to B and will collide when traveling from B to A (over this road), unless a road is always considered to have lanes for both directions (i.e. the edge is *directed* or *undirected*, respectively). On top of the basic categories, many types of graphs are defined by their structure. Sparse graphs, dense graphs, line structured graphs, mesh structured graphs are all categories based on statistical information from the graph (e.g. the ratio of vertex connections and the regularity of the structure). Such groups can be useful for improving the behavior of algorithms, at the expense of generality. For our research, we will only consider the connectivity and density of graphs.

2.1.1 Representation

A graph can represent infinite complex structures in data sets, by placing more information inside a single edge or vertex. One often used extension of complexity to a graph is assigning an intensity or weight to an edge. For example in road networks, a weight can indicate the maximum speed, or maximum throughput in cars per second. For simplicity, in this thesis, we assume the vertices to be single identifiers and the edges to only contain a source and a target vertex identifier (and have no weights).

To store data in a graph, a large range of representations can be used. The choice for a representation strongly depends on the use of the data (e.g., if the data is very dynamic, a complex representation would be impractical). The big variety of representations, can be split into two main groups: vertex-based and edge-based.

Vertex-based representation

For the implementation of graph processing algorithms, a vertex-based representation is often preferred. A vertex-based representation has to keep a list of all the information related to a vertex clustered, so the algorithms (that are often vertex oriented) have quick access to all the desired information. An example of a vertex-based representation is an *adjacency-list*, a list of vertices and the collection of neighboring vertices to that vertex (e.g. $A\{B, C\}, B\{C\}$, means edges $A \rightarrow B$, $A \rightarrow C$ and $B \rightarrow C$ exist in the network). In this representation, all edges leaving a vertex are directly accessible, hence the number of (accessible) neighbors is easily calculated. However to store undirected graphs in this representation, each edge is duplicated (since searching the adjacency list based on a target node will become significantly complex). The overhead of duplicating edges is relatively small, since this representation reduces the set of edges from the vertex, to the source vertex with a collection of target vertices (see Figure 2.2b). Various formats are based on the principle ideas of adjacency lists. For example, the adjacency array is basically an adjacency list mapped on to a 'computer-friendly' structure. Mean-

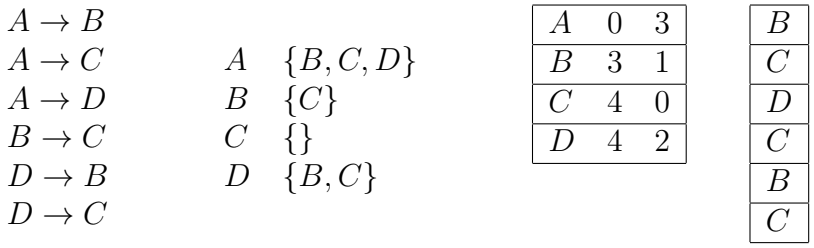


Figure 2.2: Vertex-based graph representations: (a) the list of edges between vertices, (b) an adjacency list representation, and (c) an adjacency array representation.

ing, the adjacency array removes the dynamic sizes of a vertex description – the vertex information plus adjacent vertices – which makes mapping to computer memory easier. The adjacency array uses a vector of vertices and a separate vector of adjacent vertices (i.e. a simplified edge list ordered by the source vertex that contains only the target vertices). The vector of vertices has a reference for each vertex that points to its first adjacent vertex (in the adjacency vector) and it keeps a counter of the number of connections. (This representation is used in Rodinia [45], a reference implementation used in the evaluation of our work.)

Vertex-based representations have clear structures that can help increase the performance of an algorithm. However, adjustments to the graph structure become more complex, as all affected adjacency lists have to be updated. For example, the adjacency array requires both arrays to be entirely changed whenever additional edges connect the first vertex to any other vertex.

Edge-based representation

A different approach of representing a graph focuses on the connections in the graph. Typically, such edge-based graphs contain only the list of edges in the graph, as shown in Figure 2.2a. This list of edges can be arranged according to comparative edge information, or it can contain the edges in no specific ordering. While this approach (in particular the unordered list) reveals little of the actual structure of the graph, it is very tolerant to changes to this structure, which makes it suitable for dynamic graphs.

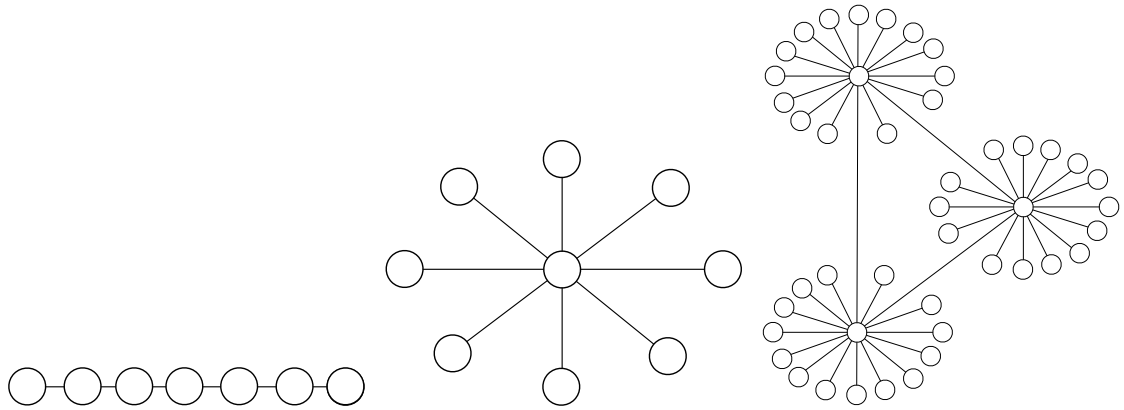
In edge-based graphs the main focus is on the relationship between nodes in the graph, i.e. information is stored on connections rather than on the vertices, hence little information is kept on actual vertices. The problem of not having information on the vertices can be solved by using both an edge list and a vertex list to represent the graph. Such hybrid solutions are used when the graph is rich in information on individual edges and vertices. For example, the Game Trace Archive [53] uses an edge list to represent an encounter in a gaming environment. Such encounters can mean various things (e.g. the players are rivals in a game, they formed a team, they had a chat conversation,

etc.) and this information is important for the context of a graph. Also such encounters can have a duration or a winner. All this information is stored in the representation of an edge. Now, for instance, if the vertices represent players, a lot of information can also be kept to store players. The Game Trace Archive uses a separate list to store the information of vertices and for the edges. From the algorithmic perspective this hybrid solution can be approached as an edge-based graph, since all the relations in the graph are stored edge-based. However, a graph can also be created without considering the edge list. For example, a graph can be constructed before any encounters occurred (i.e. no edges are yet available).
In our research, we use graphs that are represented as edge lists.

2.1.2 Data set

To provide a good overview of the performance of our algorithms, we use multiple data sets from different graphs, varying in size, structure, connectivity, and density. In this section we discuss the differences between these data sets and provide behavioral expectations of the graphs. We use graphs from the SNAP repository [32], the Rodinia benchmark [45] data sets, and statistical modeled graphs (*stats*). The SNAP repository provides subsets of real-world data sets from environments like social networks, web graphs, communication networks, and road networks. These data sets are very irregular in structure and are (in all our cases) unconnected graphs. The Rodinia data sets provide synthetic graphs, generated using random number generators. These graphs all satisfy a certain probabilistic model, that forces an average number of neighbors for the vertices in the graph, hence are regularly structured, and are connected. The *stats* graphs are graphs that, based on a model, form a theoretical upper or lower bound performance metric. All *stats* data sets are connected and have a systematical structure. In our experiments we use two statistical data sets called *chain* and *star*. These graphs represent two specific classes of graph structures that have the theoretical lower and upper bounds performance in graph traversals, respectively. The *chain* is a path that contains all the vertices of the graph and has no cycles in the graph. In other words, all vertices in the *chain* can be placed on a single uninterrupted line, as shown in Figure 2.3a. The *star* data set has an opposite structure, here all vertices have a connection to a single 'center' vertex and have no connections to other vertices (shown in Figure 2.3b).

Table 2.1 lists the graphs we use, together with some characteristic values: the number of vertices, number of edges, the average connectivity of vertices in the graph, a maximum connectivity in the graph and the diameter of the graph. The graphs vary from thousands to millions of vertices and from tens of thousands to millions of edges. The average connectivity is calculated by dividing doubled the number of edges (i.e. edges to a vertex and from a vertex both count as connected to that vertex) by the number of vertices. Based on the differences in the average and maximum connectivity,



(a) Chain graph (b) Star graph (c) Cluster of stars

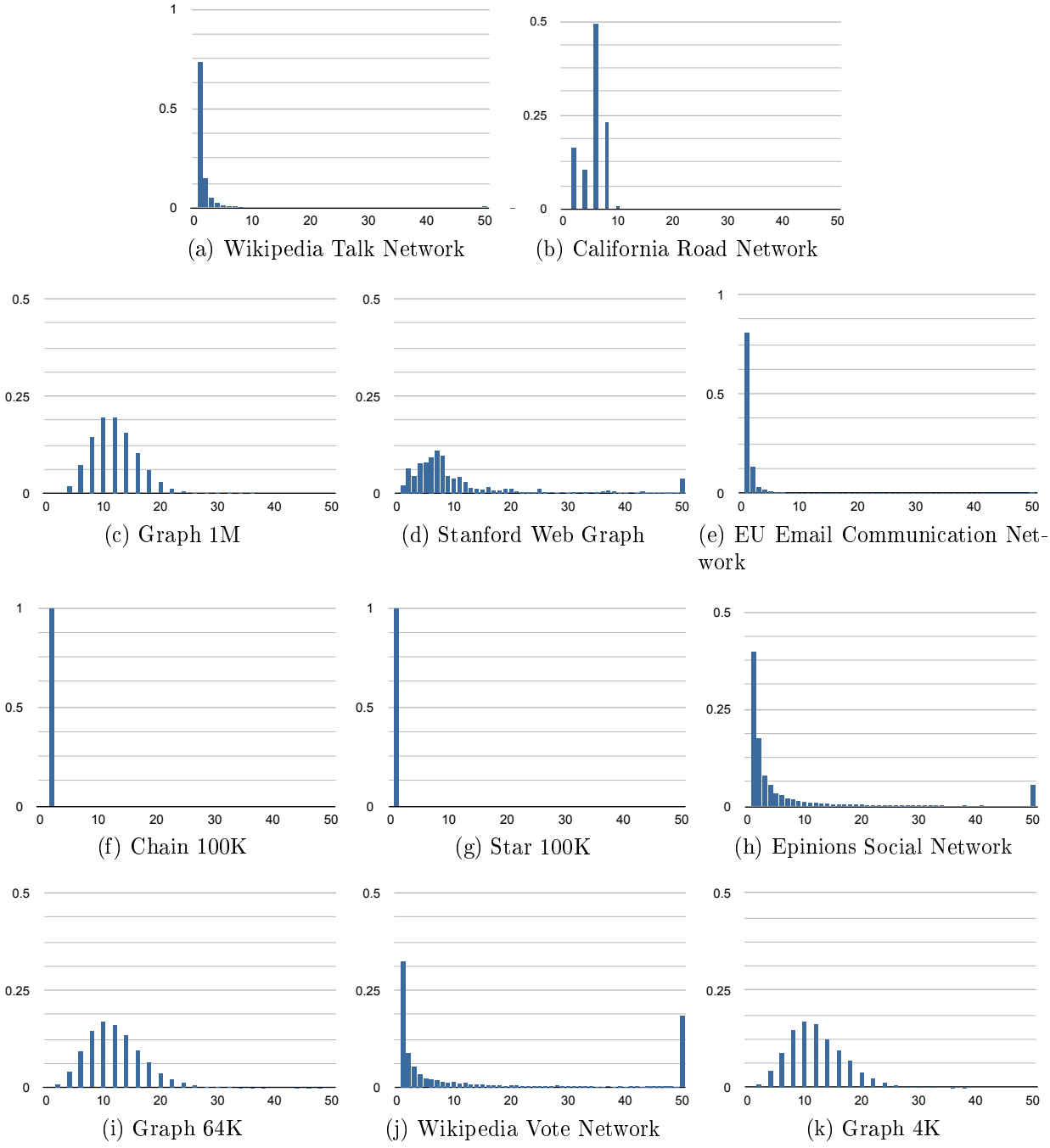
Figure 2.3: An example visualization of statistics graphs: *chain* and *star*. And a more complex graph that is more likely to occur in a network: nodes are clustered in a star-like structure.

Graph name	Name	Vertices	Edges	AVG	MAX	Repository
Wikipedia Talk Network	WT	2,394,385	5,021,410	4.19	100,032	SNAP
California Road Network	CR	1,965,206	5,533,214	5.63	24	SNAP
Graph 1M	1M	1,000,000	5,999,970	12.00	36	Rodinia
Stanford Web Graph	SW	281,903	2,312,497	16.41	38,626	SNAP
EU Email Communication	EU	265,214	420,045	3.17	7,636	SNAP
Chain 100K	CH	100,000	99,999	2.00	2	stats
Star 100K	ST	100,000	99,999	2.00	99,999	stats
Epinions Social Network	ES	75,879	508,837	13.41	3,079	SNAP
Graph 64K	64K	65,536	393,216	12.00	48	Rodinia
Wikipedia Vote Network	WV	7,115	103,689	29.15	1,167	SNAP
Graph 4K	4K	4,096	24,576	12.00	38	Rodinia

Table 2.1: Collection of data sets used in the experimentation. Where the data sets from the SNAP repository are real-world graphs, all others are synthetic graphs. With *AVG*, *MAX* be the average and maximum number of connections going to/ coming from a vertex, respectively.

it can be verified that the structure of real-world graphs is more irregular than that of synthetic graphs (varying from a few connections to a few thousand connections), with the exception of the California road network (expected since junctions in a road network can only physically support a limited number of connecting roads).

Although the average and maximum connectivity of vertices give a rough overview of the regularity of a graph, they do not provide any insight on the expected behavior



(a) Wikipedia Talk Network (b) California Road Network

(c) Graph 1M (d) Stanford Web Graph (e) EU Email Communication Network

(f) Chain 100K (g) Star 100K (h) Epinions Social Network

(i) Graph 64K (j) Wikipedia Vote Network (k) Graph 4K

Figure 2.4: Histograms of the vertex connectivity in the different data sets. Note that the connectivity is defined as the sum of incoming edges and outgoing edges for each individual vertex.

of various algorithms. In order to provide a better overview of the structure in the different data sets, Figure 2.4 shows histograms with the probabilistic values of the different connectivity counts. The histograms only show the range between 0 and 50 connections, hence the probability shown at 50 connection is actually the probability of having 50 or more connections. In Figure 2.4 some interesting characteristics can be observed. For example, the graphs from Rodinia have similar connectivity models and therefore are expected to have similar performance behavior. Note that the histograms of the Rodinia graphs only have values for even number of connections (i.e. the graphs are undirected). The graphs *Wikipedia Talk* and *EU Email Communication* have a similar histogram that indicate most vertices only have a few connections and only very few vertices have a high degree of connections. In other words, these graphs are clusters of star-like structures (see Figure 2.3c). The graphs *Epinions Social Network* and *Wikipedia Vote Network* also contain a large number of vertices with a few connections. However, there are more densely connected vertices, meaning that the clusters of vertices are smaller, and there are more connections between clusters.

Based on the observations from Figure 2.4 and the information from Table 2.1, patterns in the performance of different algorithms are expected to appear between the Rodinia data sets, between the *WT* and *EU* data sets, and between the *ES* and *WV*. Because the star structure has a theoretical upper bounded performance for graph traversals (all the algorithms in our research are based on graph traversals) and graphs *WT* and *EU* have larger clusters of star-like structures, we expect a better relative performance compared with graphs *ES* and *WV* (i.e. an increase is expected when performance is made relative to size of the graph). Similar to the Rodinia graphs, the *CR* data set is also undirected, but (in contrast to the Rodinia graphs) this graph has a small window of vertex connectivity variation (i.e. over 99% of the vertices have between 2 and 8 connections). A smaller connectivity rate means the fan out per vertex is smaller, resulting in a dispersed set of vertices, hence a larger traveling time between the ends of the graph. In other words, the relative performance of traversal algorithms for the CR data set should be worse than that of the Rodinia data sets. For the *SW* data set the histogram shows a large variety of connection counts per vertex, where 75% of the vertices have between 1 and 12 connections and little over 10% of the vertices have more than 30 connections. These statistics suggest the graph to be dense, making the traversal fast and the expected performance better than that of the Rodinia data sets.

For completeness, the *Star 100K* and *Chain 100K* histograms appear identical, but actually they represent the best and worst case expected performance. The histogram of the *ST* data set show a peak at 1 connection per vertex, which is the expected best case performance, where the histogram of *CH* show a peak at 2 connections per vertex, this is the expected worst case performance.

2.2 Programming Models

With the size of the data set increasing, the analysis complexity increases, leading to an explosion of the execution time. In particular, if the data sets tend to change rapidly in structure and size, a serious increase in execution time can be catastrophic for the analyst (i.e. before the analysis is finished the results are out-dated). By introducing parallel programming we try to reduce the execution time to make an analysis applicable to more dynamic and larger data sets.

To support parallel execution of the different parts of an analysis, we require a model to define the levels of concurrency, the communication used between concurrent components, and the dependencies between parts of the analysis. Manually applying all these design settings becomes extremely complex as the level of concurrency increases. Thus, we use a high-level programming model (i.e. a model with a high level of abstraction) to ease the design. The level of abstraction helps in writing and parallelizing algorithms, but it also introduces limitations: high-level programming models are designed to map specific instructions onto a detailed configuration in concurrent systems. As different systems require different configurations, programming models typically focus on a limited set of concurrent systems. Common used high-level programming languages are listed in [3]. For example, OpenMP and Pthread, both designed on top of the sequential C language, are used to facilitate the programming of shared memory machines (i.e., multi-core or multi-CPU). When we use accelerators such as GPUs (Graphical processing Units), CUDA [48] or CTM [21] are the languages of choice. However, these models are not only limited to GPUs, but also to a specific manufacturer: NVIDIA Corporation and ATI Technologies Inc., respectively. The OpenCL programming model [43], on the other hand, provides the option of abstracting an implementation over a larger range of hardware platforms. That is, the model allows the algorithms to be executed on CPUs or accelerators without having to alter the implementation.

In our research, we are interested in the performance impact of using accelerators for graph analysis. And since the OpenCL model allows us to have the same implementation for different hardware environments (at least at the abstraction level on which we implement), OpenCL provides the means for examining the impact of accelerators. In this section, we discuss the structure of the OpenCL model, the mapping of levels of concurrency to hardware and the memory model used.

2.2.1 OpenCL

OpenCL (Open Computing Language) is an open (royalty-free) standard for parallel programming of heterogeneous systems, managed by the Khronos Group[1]. OpenCL

[1]The Khronos Group is known for the standardization of the graphical framework called OpenGL and consists of members from companies like Altera, AMD, Apple, ARM, IBM, Intel, NVIDIA, Texas Instruments, Xilinx, which are all well-known processor vendors and/or multi-core software vendors.

supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, programming interface with abstracted hardware designs, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. [36]

OpenCL Architecture

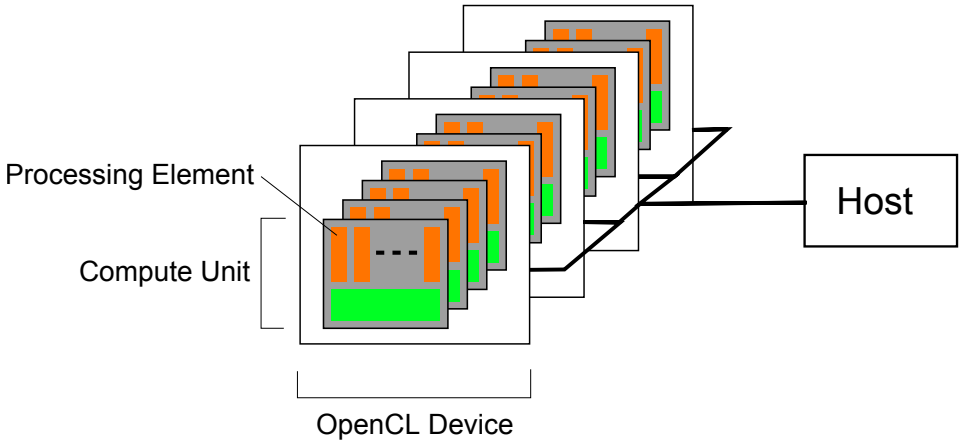
The goal of the standardization is to ultimately be able to program any combination of processors, such as CPU, GPU[2], FPGA, DSP, CellBE, using a single model. Combining this heterogeneous collection of processors into a single framework requires an architecture that is generic enough to be applicable on the different processors and at the same time is specific enough to utilize the processors for good performance. OpenCL defines a specific platform model, to which the used system is mapped. An important generalization in the model is: there must always be a control processor and one or more compute processors (host and OpenCL devices, respectively). The host, typically a single processor, initiates execution and controls the distribution of tasks / instructions over the OpenCL devices. As an example, if an OpenCL implementation is executed on a system with multiple CPUs and a GPU, one CPU can be used a host and the remaining CPUs as well as the GPU are included in performing the computations in parallel.

The platform model further divides the OpenCL devices into one or more *compute units*, which are divided into one or more *processing elements* as shown in Figure 2.5a. This model shows great resemblance with the architecture of GPUs. The GPU architecture specifies a hierarchical structure of the device having multiple streaming multiprocessors(SM), each with a set of scalar processors(SP), which are equivalent to the OpenCL device, compute units and processing elements, respectively. Hence the model is very efficiently mapped to GPUs. However, processors with different architectures experience overhead from this fine granularity of processing elements. For instance CPUs can have multiple cores. To satisfy the fine-grained parallelism of the OpenCL model, the processing elements are emulated using time-shared software blocks (called *fibers*) as seen in Figure 2.5b. These differences in the hardware architectures make designers often choose for specific processors, even though a single implementation can be used for heterogeneous systems. However, we are interested in the impacts, these differences in hardware architectures have on the performance of applications.

Application Structure

OpenCL is an extended version of C to allow parallel programming of heterogeneous architectures. In other words, an OpenCL application is written in the C language and

[2]With GPUs, we refer to General-Purpose GPUs that support use of OpenCL



(a)

Platform model	GPU	CPU	OpenCL application
Compute device	GPU	CPU	Device
Compute unit	Streaming Multiprocessors	Core	Work-group
Processing unit	Scalar Processors	Fiber	Work-item

(b)

Figure 2.5: (a) Platform model of OpenCL, with one host, one or more compute devices, each with one or more compute units, each with one or more processing elements. (b) Terminology used for different models: Platform model of OpenCL, GPU architecture, CPU architecture, and application structure of OpenCL.

contains additional instructions for initialization of processing elements and communication to the OpenCL devices. From the perspective of the application, an OpenCL environment consist of two parts: (i) a *host program*, that runs on the host and is required for the orchestration of the algorithm execution, and (ii) *kernels*, the (sub)applications scheduled by the host to run on a device. Note that multiple kernels can be scheduled to run sequentially on a single device.

Applications written in OpenCL use an index space for the collection of available processing elements and assigns kernels to a specific range of indices. Thus, an instance of the kernel is assigned to each index. Such kernel instances are called *work-items*. The index space also has a coarse decomposition into *work-groups*, where the work-items in a given work-group execute concurrently on the processing elements of a single compute unit (see Figure 2.5b for the mapping for platform terminology to that of the application). These indices (at different levels of granularity) are used by the host program to invoke processing elements and by the work-items, for instance, for data selection in data parallel programs (i.e. programs where the parallelism is realized by splitting the data sets rather than splitting the instructions). Furthermore, the indices of work-items are

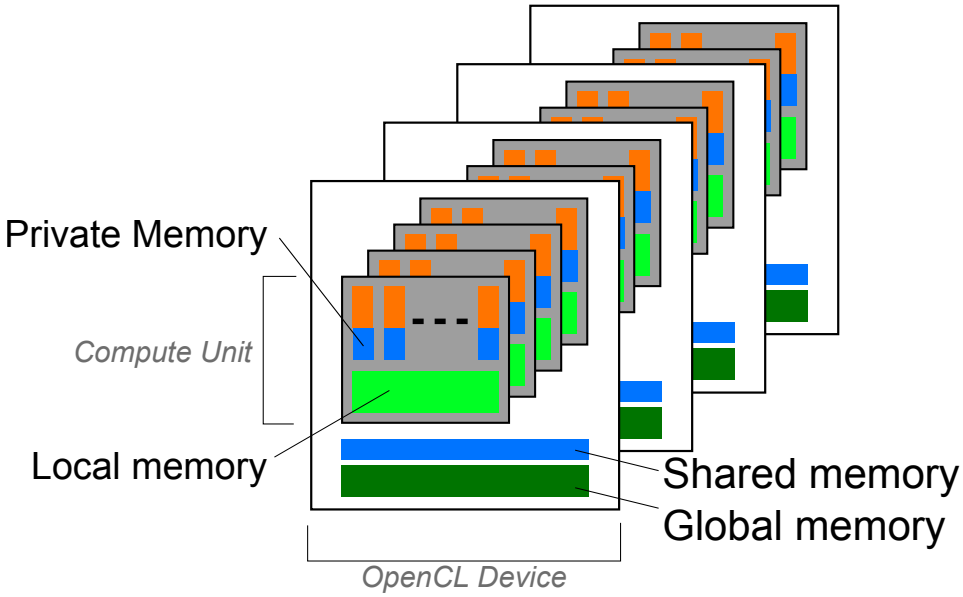


Figure 2.6: The memory structures accessible by the kernel, as specified in the OpenCL model.

used as identifiers, to provide the ability of communicating with other work-items. That is, a work-group is the collection of work-items located on a single compute unit, thus they are physically grouped together and by definition share resources and connections (see Figure 2.5a). Different work-groups can be located on different compute units and are assumed not to have any communication between them. In other words, work-items within a work-group can be synchronized, but a synchronization of all the work-items on a OpenCL device is not supported by default.

A single application consists of one or more kernels, each kernel is executed by one or more work-items. For the execution of the kernels, a queue is used on which the host program pushes the job of executing a kernel (with a specified number of work-items) and the jobs are scheduled to execute on the available work-items. This scheduling is either in-order or out-of-order (i.e. the jobs are executed in the order of queuing them, or they execute as soon as the required resources become available).

Memory Structure

In the range of OpenCL devices, many different structures should be considered by the OpenCL model. For the memory structure, a distinction is made based on the access policies of the memory. Again the OpenCL model shows great resemblance with the GPU architecture, which has memory components at each level of the hierarchical platform model (i.e., GPU, SM and SP). The OpenCL memory model, as shown in

Figure 2.6, includes four memory components: (i) global memory, (ii) constant memory, (iii) local memory, and (iv) private memory. Global memory is by far the largest in size, but also has the highest latency, while private memory is the smallest in size and has the lowest latency.

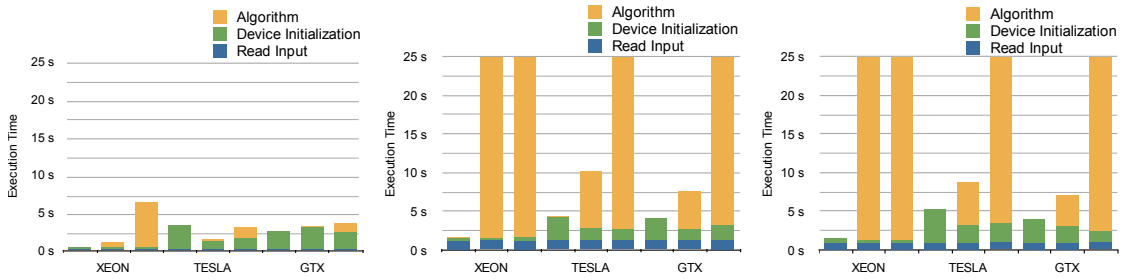
A work-item has a certain amount of private memory: typically, a set of registers that are visible only in a specific work-item. A work-item also has access to local memory, a shared memory block that is accessible by all the work-items in the same work-group. The global and constant memory is visible and accessible to all the work-items on the device, with the difference that constant memory cannot be altered by the kernel and has a smaller access time. Note that the physical locations of the memory types depend on the platform, hence the assumed differences in latencies might not hold for all platforms. For example, if the OpenCL device is a multi-core CPU, all four memory types will be located in the main memory of the CPU and the access times will (in theory) be equal. However, for generality reasons, the OpenCL model requires an explicit specification of the used memory types. Another important difference between the memory types, is the accessibility of the memory by the host. The host can only read and write data to the global and constant memories.

2.2.2 Using OpenCL

In our research, we employ OpenCL for the comparison of graph analysis on homogeneous systems – systems with multi-core CPUs – and heterogeneous systems – systems using GPU accelerators and a CPU host. For the implementation of these algorithms, we use a single type of kernel that follows a data parallel structure, that is, the parallelism of the algorithm is exposed by concurrent data evaluation rather than by splitting the work in different tasks.

In Table 2.3 can be observed that the data sets have a significant number of vertices and edges. To place the initial edge-list data set on the memory of the OpenCL device, we require at least two integer values (i.e. the source and destination identifiers) for each edge and, for a result, at least one integer value is required for each vertex. We choose to place this large chunk of edge information on the global memory of the GPUs, leaving the local memory and private memory available for storing intermediate values of the algorithms.

To perform our experiments, we use a wide-area distributed system, called DAS-4 [18]. A collaboration between a number of Dutch universities host the DAS-4 system. The goal of DAS-4 is to provide a common computational infrastructure for researchers. The DAS-4 system has support for various programming model, including OpenCL, and it allows us to test different systems (different GPUs and multi-CPU systems).



(a) 7K data set (b) 64K data set (c) 74K data set

Figure 2.7: The total execution times of three data sets, each ran on three different parallel systems, with three different algorithms. Note that for clarity the execution times only show up to 25 seconds.

Execution time as metric

We use OpenCL for our research, because it allows the mapping of one algorithm onto various parallel systems. One of the costs of having a parallel system is the part before the actual execution, where the initial data must become available to each individual processor. In OpenCL, the model specifies use of a host and one or more devices, so the information needs to be copied from the host to the devices. Also, the devices require to be initialized/ reserved to ensure the device to be ready to execute the task at hand.

So, we divide the complete execution of an algorithm on a parallel system into different parts:

1. Reading the input, which refers to reading the data set that is stored in the system. We consider this as an isolated part (i.e. a so called 'one-time overhead'), as it is a constant part of the execution (i.e. it can be read once for execution of multiple algorithms). We note, however, that this can become a serious bottleneck for very large graphs. Despite this, optimizing this process is out of the scope of our research.

2. OpenCL initialization, which includes a combination of the initialization of the OpenCL framework, the preparation of the devices, copying the initial values to/ from the various devices, and releasing the reservation of the devices after the execution is finished.

3. Kernel execution time (or algorithm time), which is the actual execution of the algorithm at hand. In Figure 2.7, the impact of the three different parts on the execution is shown for three sizes of data sets (the three sub-figures) and three different algorithms. Note the differences in the reading and initialization, and their relation to the algorithm (grouped per system). For clarity, the execution

times in the figure are limited to 25 seconds. Figure 2.7 shows that the overhead of OpenCL and the reading of the input are large for simple algorithms, but less important with respect to the overall performance for larger data sets and more complex algorithms. Also note that there is a significant difference for the device initialization of the various systems (the initialization of the Intel Xeon takes significant less time that the GPUs, where the Tesla is slightly faster in initializing than the GTX).

The total execution time is a relevant metric for the end-user, but it includes the large OpenCL overhead; therefore, when we look to understand the impact of the accelerating algorithms with a GPU, we will not look at the total time, but rather at the kernel time. The kernel time is the actual time the algorithm requires to finish, and it is easier to analyze in isolation. This will be used along this thesis as our main performance metric.

Chapter 3

Related Work

In this chapter, we discuss existing related work, starting with related research on graph analysis, then work on parallel programming is addressed, with the extension of graph analysis on parallel systems. We end the chapter with an overview of work on accelerators (again extended with graph analysis on these systems).

Many studies use graph analysis to find patterns in data. In [30,35], the authors were interested in finding paths through various networks (i.e. mazes or electrical circuits, respectively) and, in particular, in finding the shortest paths. These studies are generic, in the sense that they consider all connections to be equal (i.e. breadth first search, BFS). In [2, 5, 13, 41, 50], the path finding is more specific for a type of graph (i.e. a graph in which connections are weighted), but they still find a path using a single starting point (i.e. single-source shortest path, SSSP). This approach is also used for image processing, where the image is represented as a graph of pixels and the graph analysis forms decision models, used in an image segments recognition, e.g. [6, 20, 33].

Other studies specifically focus on social networks, and in particular on the influences of individuals within these networks (BC) [7, 11, 14, 39]. Based on the position of individuals in a social network, a centrality value is calculated and used as a hypothetical influence rate within a group. Which in turn is based on an extended version of the path finding (APSP), where all shortest paths in a network are considered to provide routing information [12, 23].

These studies on shortest paths (BFS and SSSP), all-pair shortest paths (APSP) and centrality, prove there is a wide range of applications for graph analysis. Also, they represent different gradations in time-complexity and algorithmic-complexity, while having great resemblance (i.e. centrality is based on an all-pair shortest path operation, which in turn is based on finding shortest paths). In our research, we are interested in this resemblance and we try to extrapolate a performance boundary for the more complex analysis using that of the simplistic analysis. Specifically, we exploit the resemblance of these algorithms by reusing the most simple analysis (BFS) in the construction of more complex algorithms (APSP and BC). This 'iterative' approach of increasing the

complexity of the analysis allows us to study the scalability of the system (hardware, software and data) with regard to different parameters.

As graph problems grow larger in scale and more ambitious in their complexity, they easily outgrow the computation and memory capacities of single processors. Given the success of parallel computing in many areas of scientific computing, parallel processing appears to be necessary to solve the resource limitations of single processors. Unfortunately, the algorithms, software, and hardware that have worked well for developing mainstream parallel scientific applications (like the algorithms used in [9, 33, 52]) are not necessarily effective for large-scale graph problems. Because graph problems have some inherent characteristics (like data-driven computations, irregular structures, and poor locality, listed in [4]), they are poorly matched to current computational problem-solving approaches. Therefore, work a parallel graph processing stands out by its specific approaches.

For example, in [51], an attempt is made to improve the throughput of graph searches on parallel systems by data partitioning and message compression. They suggest a 2D partitioning scheme, that increases throughput while maintaining the scalability of the system, and applying this to a BFS. In [31], the authors propose to replace the queueing of reached positions in the graphs (also in BFS) with a more efficient parallel approach called bags – a mechanism that manages splitting and merging of queued operations such that they can operate in parallel. For a centrality algorithm, [25] describes a different technique to gain efficiency in parallel systems: they replace the SSSP algorithm, that is used to calculate the pair-dependencies for all pairs, with a BFS algorithm (as suggested in [7]). But they track the successor nodes instead of the predecessor nodes – to compute the centrality values, back-propagation is used for the accumulation of the influential ratio on various paths –, which turns out to have a significant influence on performance. These algorithms and frameworks depend on an underlying programming model. In [22], a comparison is made between two common used models for multi-cores, that show the importance of understanding and tuning the programming model, based on the available parallel system and application.

In recent years, the field of parallel computing is extended by an increasing interest in the use of accelerators, due to their high performance and high power-efficiency (i.e. they consume relative little power with respect to their computational power). Until recently, the capacity of GPUs was not considered sufficient for general purpose scientific computing, but the rapidly growing technology of graphical cards changed this view. Current research more often focuses on the use of GPUs (i.e. General-Purpose GPUs) to accelerate general purpose scientific computing, e.g., [9, 42, 44, 45]. In [42, 44], a comparison is made between the performance of multi-core CPU and GPU for various algorithms, where the GPU shows promising performance for the larger data sets.

The authors of [46], [29] and [24] push this comparison a step further by looking at multi-CPU, GPU, CellBE, and FPGA (i.e. two additional hardware accelerators). Unfortunately, for the latter two accelerators only little research is conducted, hence programming models and reference implementations are scarce, making the comparison limited and difficult to verify.

On top of the mentioned research on accelerated scientific computing, for a few years, graph analysis is subjected to accelerators. In [34], an overview is sketched on the challenges of adapting graph analysis to heterogeneous systems. In [19,49], comparisons are made between graph analysis running on a single CPU (i.e. running sequentially) and those accelerated with a GPU. For these comparisons, different graph analysis algorithms are used, for which the GPU shows significant performance improvement. In [26, 47], efforts are made on adapting graph algorithms to the hardware structure of GPUs: the instructions and the data are ordered such that they better fit the GPU architecture, without any loss of correctness.

A large portion of the research on accelerated graph analysis, focuses on specific cases where they aim at improving an algorithm for specific environment. However, for our research, we are interested in a solution that gives a good estimate on the impact of using accelerators, without limiting us to a single hardware platform (or single accelerator, for that matter).

With the increasing data sizes and computational complexity, parallel computing helps in reducing the execution time, but brings along additional complexity in programming, which leads to an urge for generic frameworks for assisting in the development of parallel (graph) algorithms. For example, Parallel Boost Graph Library (Parallel BGL, formerly known as the Generic Graph Component Library [28]) is a library of parallel graph algorithms and data structures. In the attempt of making a generic framework for programming graph algorithms, many studies extend and improve this Parallel BGL framework, e.g., [4, 17, 37, 38].

Another approach is to assist the programmer by introducing abstraction levels (i.e., use higher level programming models). A widely studied programming model is the Bulk Synchronous Parallel model (BSP) [10, 52]: a bridging model for parallel computation that uses a step-wise execution of algorithms, with a synchronization layer between the different steps. This model is specialized for graph processing in: CGMGraph [8], Pregel [15], MEDUSA [54], and TOTEM [1]. CGMGraph, is a library of graph algorithms designed in a message passing environment. CGMGraph divides the algorithms in computation steps, and uses a synchronization layer between these steps to handle message exchanges. This message passing is required because algorithms are performed on distributed data, located on an arbitrary number of processors. Pregel uses a similar message passing technique, but more effort is put in addressing fault tolerance: with the introduction of checkpoints, a long running algorithm does not have to start from

scratch in case of system failures (in a cluster). Also, the authors put more emphasis on the fact that the distribution is not necessarily only data-driven, but can also be driven from the computational perspective. From the approach of Pregel, the authors of [54] designed a graph processing framework, called MEDUSA, that emphasizes the use of one or more GPU accelerators. TOTEM is a slightly different framework which allows GPU accelerators to assist in the graph processing, rather than the graph processing being completely preformed on GPUs (as is the case in MEDUSA).

In our research, we merely focus on the performance of accelerators in graph analysis. For now, we use custom implementations of graph algorithms to compare the scalability of the performance on more fine-grained hardware and more complex algorithms. In our research we encountered limitations to this approach, hence a logical next step for our future research would be to add dedicated graph processing programming models to build a more thorough comparison. For example, the MEDUSA or TOTEM models show promising features for using multiple accelerators for a single graph analysis.

Chapter 4

Breadth First Search

Graph analysis often requires the retrieval of characteristics of a provided graph (e.g., the density of the graph). By traversing through the graph in a structured manner, algorithms are able to retrieve viable information with relative little effort, from the point of view of the analyst.

Graph traversals are used in computer science for solving a big variety of problems. Due to their systematic approach of visiting all vertices and performing computations on each of them, a single algorithm could be used for a large set of different applications.

Widely studied graph traversal algorithms are *Breadth First Search* [30,35] and *Depth First Search* [40]. Possible uses of such algorithms include finding the distance of a vertex to any other vertex in the graph or generate the minimal spanning tree [41] from the source graph. Characteristics as such could help the analyst in finding an appropriate algorithm to further analyze the graph. However as graphs tend to grow large, traversal algorithms become very time consuming, and are likely to outgrow computation and memory capacities of traditional processors [4]. In this chapter we present an approach to implement a *Breadth First Search*, a parallel implementation programmed using OpenCL, and a study on the impact of the Graphical Processing Unit (GPU) on the BFS performance.

4.1 Sequential Breadth First Search

Studies on the Breadth First Search (BFS) started 50 years ago with the independent discovery by Moore [35] and Lee [30], while studying the problem of finding paths through mazes, and routing wires on circuit boards, respectively.

The strength of the BFS lies in its simplicity, as it is one of the simplest algorithms for searching a graph. The BFS explores a graph level by level. Given a graph $G = (V, E)$, where V is the collection of vertices and E is the collection of edges in the graph, there is a source vertex s which forms the top level (level 0). Now BFS traverses all edges

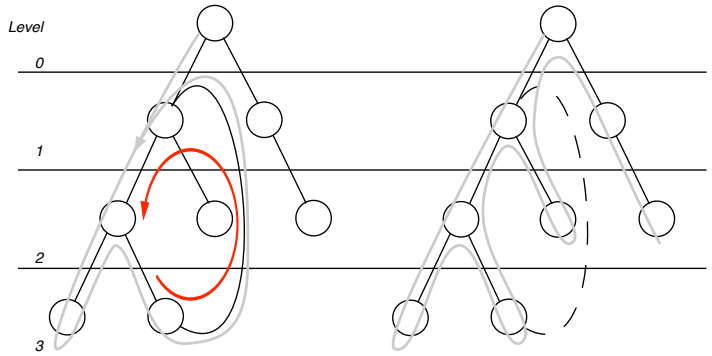


Figure 4.1: Breadth First Search traversal sequence, loop problem solved using vertex coloring.

outgoing from vertex s and places the destination vertex in level 1 (i.e. $V_{level1} = \{v \in V | e_{s \to v} \in E\}$). Next it traverses all outgoing edges from each of the vertices at level 1, to place the newly discovered vertices at the next level. The algorithm finishes when no new vertex is discovered. In a general sense breadth first search systematically explores edges outgoing from vertices at level i and place the destination vertex in level $i + 1$, given that the vertex has not yet been discovered in prior levels, resulting in a tree of all the (in)directly connected vertices of s. Note that, the algorithm requires some coloring to distinguish *discovered* vertices from *undiscovered* vertices to prevent infinite loops as illustrated in figure 4.1.

Because of the continue need for characteristic analysis on graphs and the relative straightforward nature of the BFS, a lot of effort has been put on enhancing this algorithm. Prim's minimum-spanning-tree algorithm [41], Dijkstra's single-source shortest-paths algorithm [37,50] and the Bellman-Ford algorithm [5,13] are widely known examples of algorithms with ideas similar to those in BFS. Many other studies apply changes to the structure of the BFS in order to make the search dedicated to the field of interest [20]. This wide range of research merely illustrates the adaptability of BFS. For the sake of clarity, only the general BFS algorithm, as listed in Algorithm 1, is considered here.

Edge-based Breadth First Search

Thus far the discussion has been focussed on how the BFS algorithm traverses the graph, neglecting the fact that graphs can be represented in various ways. The majority of the above mentioned research on BFS assumes the graphs to be represented in a vertex-based adjacency list, since this is considered to be ideal with respect to algorithmic

Algorithm 1 A general implementation of the Breadth First Search, graph $G = (V, E)$. Using a graph that is represented in an adjacency list. Where s is start vertex, N is number of vertices, M is number of edges

```
1:  function BFS_AdjacencyList(V, s)
2:      Q ← ∅
3:      for j = 1 → N do
4:          V_j ← UNVISITED
5:      end for
6:      add(Q_0, V_s)
7:      V_s ← VISITED
8:      i ← 0
9:      while Q_i ≠ ∅ do
10:         for v ∈ Q_i do
11:             for e ∈ v.edgeList() do
12:                 if getSource(e) = UNVISITED then
13:                     add(Q_{i+1}, getDestination(e))
14:                     depth[getDestination(e)] ← depth[v] + 1
15:                     inc(v, out_count)
16:                     inc(getDestination(e), in_count)
17:                     getSource(e) ← VISITED
18:                 end if
19:             end for
20:         end for
21:         i ← i + 1
22:     end while
23: end function
```

complexity and size. But a graph can also be represented as an unordered edge list (as discussed in section 2.1.1). In this case, algorithms are likely to require additional pre-processing steps on the input graph. As graphs (e.g. graphs of social networks) are rapidly growing in size and density, this pre-processing is no longer a one time cost, making edge-based algorithms interesting for our research.

When shifting from a vertex-based BFS (Algorithm 1) to an Edge-Based BFS (Algorithm 2), the following graph property suggests a degradation of the performance:

- For any *connected* graph, $M = xN, x \geq 1$, M=number of edges, N=number of vertices. With one exception, namely a graph that is a single line of vertices ($M = N - 1$).

Even so, this approach can be very useful, due to its potential of having a high level of parallelism as we will show in the next section.

Algorithm 2 Edge-based Breadth First Search implementation, graph $G = (V, E)$. With start vertex s, the number of vertices N and the number of edges M

```
1: function BFS_EdgeBased(E, s)
2:     Q ← ∅
3:     for j = 1 → M do
4:         E_j ← UNVISITED
5:     end for
6:     add(Q, V_s)
7:     changed ← 1
8:     while changed = 1 do
9:         changed ← 0
10:        for e ∈ E do
11:            if e = UNVISITED then
12:                if getSource(e) ∈ Q then
13:                    add(Q, getDestination(e))
14:                    e ← VISITED
15:                    depth[getDestination(e)] ← depth[getSource(e)] + 1
16:                    inc(getSource(e), out_count)
17:                    inc(getDestination(e), in_count)
18:                    changed ← 1
19:                end if
20:            end if
21:        end for
22:    end while
23: end function
```

In a BFS that accepts an edge list as input, the iteration over the entire set of edges is required. By a simple check on the source vertex of an edge, the algorithm can determine which edges to traverse, hence which vertex to place in the next level of the resulting tree. Or more generally, $\forall e \in E | e_{source} \in Q_i$ add $e_{destination}$ to collection Q_{i+1}, where E is the collection of edges in the graph and Q_i is the collection of queued vertices at level i. Note that coloring is still required to prevent infinite loops.

A first expected degradation in performance can be found in the first part the algorithm, i.e. the initialization of the labels (coloring). Second, in the body of the loop which systematically traverses the graph, a similar performance degradation is expected to arise.

4.2 Parallel Breadth First Search

Both Algorithm 1 and Algorithm 2 are designed in a sequential fashion, i.e. a single processor is used to perform all operations in order, starting at line 1. For this reason, even though Algorithm 2 has a larger number of iterations, increasing the speed of the processor executing the algorithm can lead to a shorter execution time compared to that of Algorithm 1. One might argue whether this is still a fair comparison, but this merely illustrates the high correlation of the performance of a single processor with the speed of a sequential algorithm. In order to outgrow this limitation, the introduction of additional processors, to split the workload, should result in a reduction of the overall execution time.

The BFS algorithm requires a list of vertices to discover neighbors for the next level. However, the vertices in the queue of level i are not known before level $i-1$ is processed. Hence, when splitting the workload over an arbitrary number of work-items, communication and synchronization are required between levels. Relaxing the synchronization between levels leads to an improvement of the execution time, but will also violate the basic BFS specifications from [30, 35]. For example, there are two work-items WI_1 and WI_2, where WI_1 is slightly faster than WI_2. When WI_1 finished processing its share of the workload for level i, it will start at level $i+1$ while WI_2 is still processing level i. If WI_1 now discovers a node X it will place it on level $i+1$, but there is the possibility this should actually be a node at level i that still had to be discovered by WI_2. Although this graph traversal is not a traditional BFS, it is capable of retrieving certain graph invariants, like in- and out-edge ratio or connected component sizes. Due to less synchronization, this relaxed version of BFS is expected to traverse the graph faster than a traditional BFS. In the experimental section we will evaluate the performance of this *Relaxed-BFS*.

Various techniques can be applied to implement synchronization. We distinguish here two types: (i) host-based synchronization, where the host assigns a single level of the BFS to work-items, after the work-items are done, it determines if a next level is required, and assigns the work-items to the next level or finishes the algorithm, and (ii) kernel synchronization, where the host assigns the entire task to work-items, hence communication is required between different work-items to determine when to continue to the next level in the algorithm [34]. Since no additional reasoning is required in between two levels, kernel synchronization is a suitable technique for our implementation. This requires communication between the different work-items, to wait for all the items to finish the current level and some controls that determine whether a next level exists (since this is not known a-priori). These next level control signals also exist in the sequential version, only now they have to be communicated throughout the system.

To facilitate the different work-items with an equal amount of work, we need to analyze the algorithm and find chunks of operations (i.e. kernels) that can be executed

concurrently, with the least amount of dependencies between them. Our choice is to split the edge list over the work-items since edges have no direct dependencies between one another. The iteration over edges is the core part of the BFS algorithm and has to be performed on each level of the BFS. As a result, there is a high correlation between the performance of BFS and the diameter of the data set (i.e. the maximum number of levels required for a BFS).

As Algorithm 3 illustrates, BFS is a memory bound algorithm (the body of the algorithm mainly consists of memory read and write operations rather than arithmetic operations).

Because all work-items use shared memory (i.e. a shared queue with all the vertices of the current level and a queue for the next level), accesses to the memory must be controlled to ensure correctness, meaning only a single work-item can read or write a memory block at the time to prevent overwriting of data. The use of atomic operations can ensure such mutual exclusiveness, where atomic operations enforce the work-items to wait for exclusive rights for reading or writing to a single memory block. However, since the goal of the traversal is to retrieve viable information from vertices in the graph (e.g. depth in tree, in- and out-edge ratio), such information is kept in the structure. Because a structure is likely to be located on various blocks of memory, atomic operation are not sufficient to enforce mutually exclusive rights. For this we use additional signals (locks) to indicate if the structures are used by other work-items. Because the lock signal can be controlled using atomic operations, the use of such a locking mechanism still ensures mutual exclusiveness.

Algorithm 3 describes our parallel BFS that uses synchronization barriers and exclusive memory operations, where "$atom\_xchg(a, x)$" is an atomic exchange between variable a and x, "$atom\_cmpxchg(a, x, y)$" is an atomic compare/exchange which compares a is equal to x and, if true, exchanges y with a, and "$barrier()$" is an memory barrier to synchronize the BFS levels (for a more extensive description of these operations, please check the OpenCL guide [36]). Note that the locking mechanism is applied with the atomic operations on lines 8 and 14.

OpenCL Utilization

The OpenCL architectural model, as described in section 2.2.1, has two distinctive levels of parallelization. In local parallelization, an arbitrary number of work-items in a single work-group are used, these can communicate to each other and share the work-group resources. In global parallelization, a list of work-groups is used; these are physically located on different parts of the device. Both the work-items and the work-groups work in parallel to complete the task at hand. Our parallel BFS implementation requires communication between work-items for synchronizing between the different BFS levels. Since the OpenCL model does not allow direct communication between work-groups, only local parallelization can be applied to this implementation, which is a major lim-

Algorithm 3 Parallel BFS implementation in OpenCL, for graph $G = (V, E)$. Where s is start vertex, $E_{thread} \subseteq E$ equals the edge list assigned to this thread (PU).

```
1: function KERNEL_BFS(E, s)
2:     Q_i ← {V_s}
3:     hasNextLevel ← 1
4:     i ← 0                                                    ▷ Set Current Level
5:     while hasNextLevel = 1 do
6:         for e ∈ E_thread do
7:             if e = UNVISITED & getSource(e) ∈ Q_i then
8:                 while (atom_cmpxchg(getDestination(e) → locked, 0, 1))   ▷ Lock
9:                 add(Q_i, getDestination(e))
10:                e ← VISITED
11:                depth[getDestination(e)] ← depth[getSource(e)] + 1
12:                inc(getSource(e), out_count)
13:                inc(getDestination(e), in_count)
14:                atom_xchg(getDestination(e) → locked, 0)          ▷ Unlock
15:                atom_xchg(hasNextLevel, 1)
16:            end if
17:        end for
18:        i ← i + 1                                             ▷ Increase Level
19:        barrier()                                  ▷ Level Synchronization
20:    end while
21: end function
```

itation to the degree of parallelism. For the synchronization steps in the prior BFS, we use build-in mechanisms called *barriers* (i.e. a barrier is an operation that forces each work-item within a group to hold until all work-items have reached it). However because these barriers are only possible within a single group, adding groups would lead to partial synchronization, hence potential violations of the BFS specifications.

To solve the synchronization between groups, we propose an extended barrier mech-

Algorithm 4 Inter-group synchronization signaling.

```
1: barrier()
2: if get_local_id() = 0 then
3:     atom_inc(inter_SM)
4:     while (atom_cmpxchg(inter_SM, group_count, 0) ! = 0)
5: end if
6: barrier()
```

anism, illustrated in Algorithm 4. By using inter-group synchronization signals on the

Type	Name	Clock Frequency	Memory Size	Memory Bandwidth	Multiprocessors
CPU	Intel Xeon CPU E5620	2458 MHz	24 GB	25.6 GB/s	16
GPU	Nvidia Tesla C2050 / C2070	575 MHz	3 GB	144 GB/s	14
GPU	Nvidia GeForce GTX480	700 MHz	1.5 GB	177.4 GB/s	15

Table 4.1: Specification of hardware platform used in the experimentation.

global memory of the device, we are able to emulate communication between different groups. However, since the number of work-items can be over a thousand per group, raising the inter-group signals is likely to cause serious delays. Therefore, we decided to combine the existing barrier mechanism with the inter-group signals, resulting in a single outgoing signal per group after a first barrier and a second barrier to wait until all groups raised their inter-group signal. This approach requires a single work-item per group to be responsible for the inter-group signal and verification of the groups raising their signal (we use the build-in function *get_local_id()* to make sure only a single work-item will claim this responsibility). In the experimental section of this chapter (section 4.3), we show that using more groups to perform a single BFS has a significant performance impact.

4.3 Experiments and results

For the evaluation of the BFS implementations introduced in this chapter, we require a hardware environment supporting parallel executions. For the sake of clarity, we choose to use a single type of CPU for the sequential algorithm, the CPU-based parallel implementation and for the host of the GPU-based implementation. We perform our experiments on the DAS-4 system, we use this system because it enables us to run tests for different configurations under similar circumstances (i.e., DAS-4 allows for isolation of tasks, avoiding situations where machines would be running different task). On the system we ran tests using the configurations listed in Table 4.1. Note that both Nvidia Tesla and Nvidia GeForce configurations use as a host processor the Intel Xeon CPU.

Data sets

As mentioned in the previous section, BFS is a data-dependent algorithm, i.e. graph properties like the number of edges, the number of vertices, diameter, and in/out edge ratios significantly influence its performance. In our experiments we use two kinds

of graphs, (i) synthetic graphs, i.e. randomly generated graphs with predetermined properties like probabilistic in/out ratios and (ii) real world graphs, i.e. subsets of existing networks like road networks, social media networks, and email exchange networks. Specifically, in the experiment we use three synthetic data sets from the Rodinia benchmark [45] and six real world data sets from the SNAP repository [32]. On top of these, we examine two synthetic data sets based on statistical models that reflect the theoretical worst case and best case performance of the BFS: a chain of vertices where all but the first and last vertices have two neighbors connected and a star – one center vertex with $n - 1$ connections and all other vertices with a single connection to this center vertex.

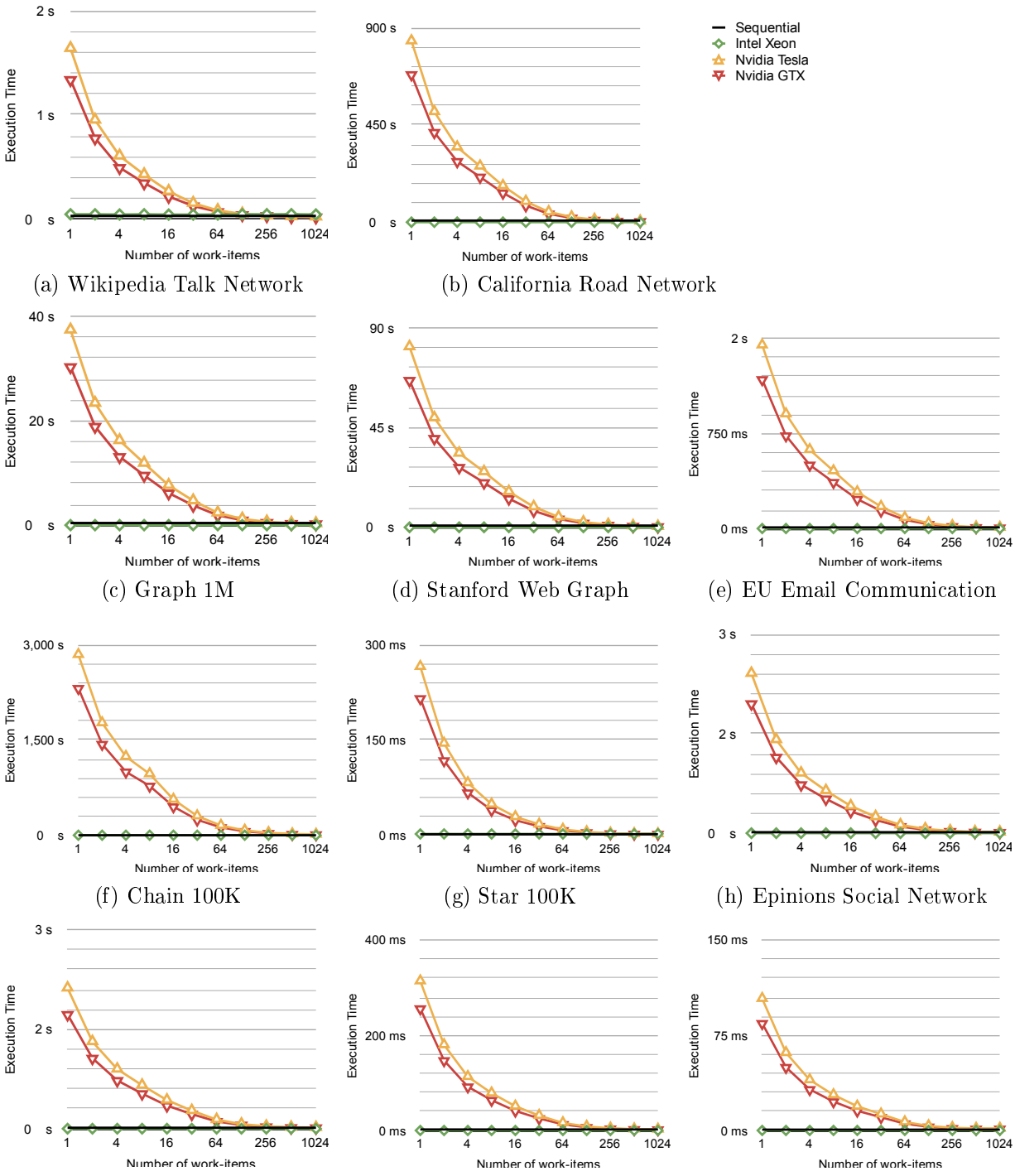
Local parallelization

First, we present an experiment based on Algorithm 3. In this experiment we evaluate the performance impact of distributing the BFS computation over an arbitrary number of work-items located in a single work-group. At this level of parallelization, work-items have shared resources and communication between each other, making atomic operations sufficient for ensuring correctness of the results (as described in section 4.2). Recall that in the CPU architecture such a fine-grained level of parallelism is not supported (section 4.2), hence the multi-core CPU implementation uses (time-shared) fibers to emulate parallelization and the performance is expected to be (almost) constant for this experiment.

Figure 4.2 shows the performance of BFS for the different data sets, when varying the number of work-items (increasing logarithmic from 1 to 1024). The presented execution time is the median value of a total of 15 runs, to eliminate any irregular behavior (all these times are kernel execution times, see Section 2.2.2).[1]

From Figure 4.2, we can clearly see the difference in processor complexity: the individual processors in GPUs are much more simplistic and less powerful than that of the CPU. The figure also shows is a significant performance improvement for the GPUs with the introduction of more work-items. However, for most data sets, the GPUs only approaches the time of the sequential algorithm (not clearly visible in this figure, but this will become more clear in the next experiments). As expected, Figure 4.2 shows an almost constant execution time for the Intel Xeon CPU, for all data sets. Furthermore, the execution times for the CPU, both sequential and parallel, indicate similar performance (in this figure close to the zero lines, due to the large execution times of the GPUs with a few work-items). We can conclude that the use of fibers has insignificant influence on the performance in this experimental setup.

[1]Because of the large variety of size of data sets, there is also a large difference in execution times show in the different sub-figures.



(a) Wikipedia Talk Network (b) California Road Network

(c) Graph 1M (d) Stanford Web Graph (e) EU Email Communication

(f) Chain 100K (g) Star 100K (h) Epinions Social Network

(i) Graph 64K (j) Wikipedia Vote Network (k) Graph 4K

Figure 4.2: Performance of local parallelization using OpenCL, with increasing number of work-items. Note that the time scale differs significantly between data sets.

Global parallelization

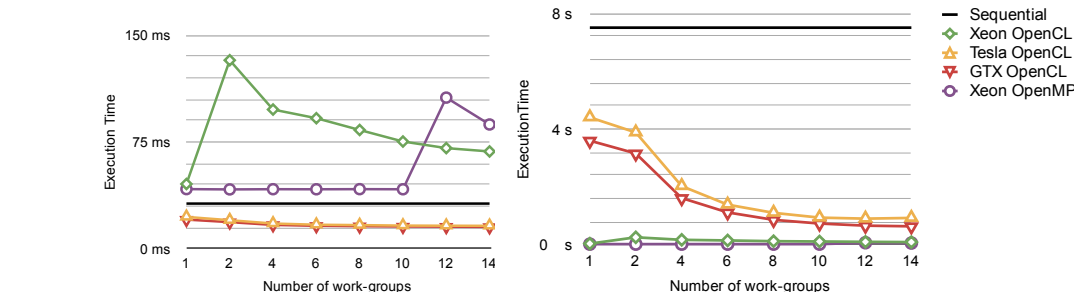
To further increase the parallelism available for the BFS, we make use of multiple work-groups (as described in section 4.2). Since the maximum number of work-groups varies between our three test environments (listed in Table 4.1 are the maxima for each). We use a range from 1 to 14 groups (to accommodate the smallest platform). Each group consists of 1024 work-items for both GPU implementations and the Intel Xeon CPU (as concluded from Figure 4.2, varying the number of work-items per work-groups has no performance impact on a CPU).

Figure 4.3 shows the performance of BFS for the different graphs from Table 2.1. We plot the execution times of our implementations (CPU and two GPU implementations) while increasing the number of used work-groups. Note that here, we also show the performance when using the OpenMP library [27] to parallelize our BFS implementation, as well as the sequential execution times.
As the sequential execution does not use any threads, this reference line (the black line in the plots) is merely used to illustrate the performance gain/ loss for the parallel implementations. In Figure 4.3, some unexpected behavior appeared. For the Intel Xeon CPU, the execution of the BFS on a single work-group is in most cases faster than the execution using multiple groups. We expect this to be the cause of memory access patterns: since BFS is data dependent and more parallelism leads to more arbitrary data requests, introducing parallelism can degrade performance. The implementation using OpenMP shows a sudden peak at 12 work-groups, it is unclear to us what exactly happens here. But the figure also shows the performance improvements of GPUs with more work-groups. In some cases, the GPUs can even outperform the parallel CPU version. However, in general, this application seems not to have extensive profit from parallelism (i.e. the parallel implementations do not perform significantly better than the sequential).

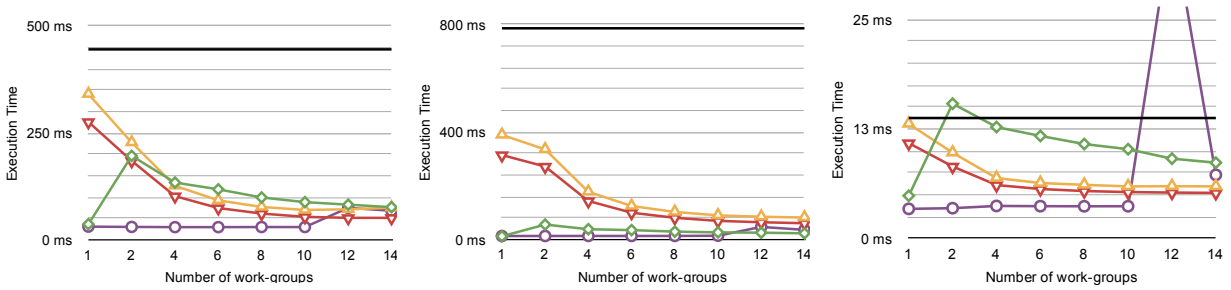
Random input

In Chapter 2.1, we claim that graphs (in particular large-scale social graphs) are rapidly growing in size and density. This growth consists of introduction of additional edges to the edge list, resulting in an unordered list of edges. In other words, the list of edges that represents the graph has no systematic ordering, with respect to source or destination vertices. To evaluate the impact of such an unordered input, we execute our algorithm using different forms of orderings of the edge lists.

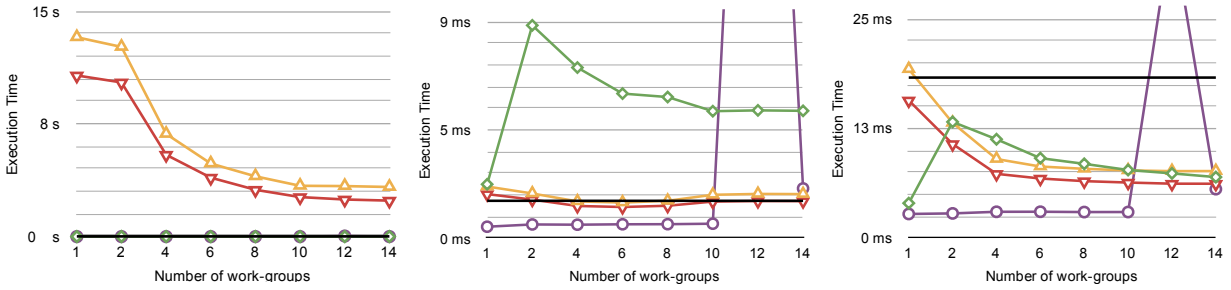
First we measure a reference execution time on edge lists that have a specific ordering: based on the source of an edge, and on the destination of an edge. Further, we use a set of 10 different shuffles for a more accurate analysis. Of this set of 10 shuffles, 9 use an ordering generated using a pseudo-random generator (the default random number generator implemented in the C language). For the last data set we try to minimize the



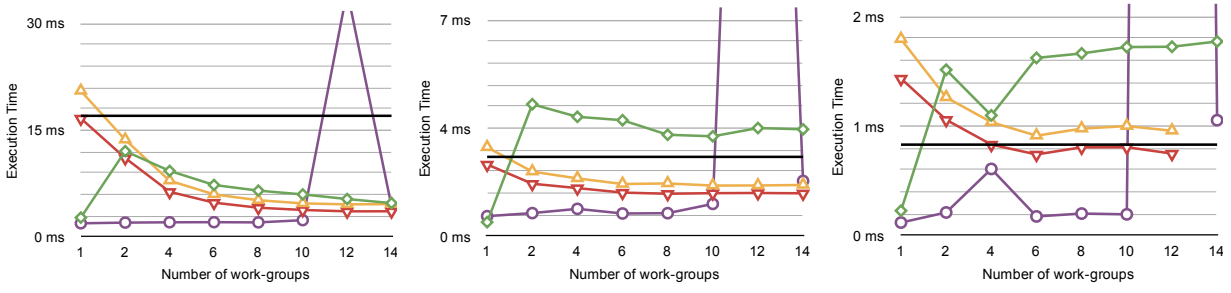
(a) Wikipedia Talk Network (b) California Road Network



(c) Graph 1M (d) Stanford Web Graph (e) EU Email Communication Network

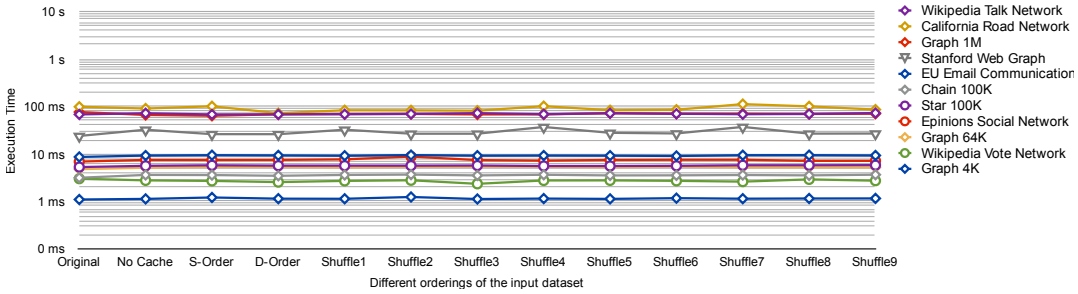


(f) Chain 100K (g) Star 100K (h) Epinions Social Network

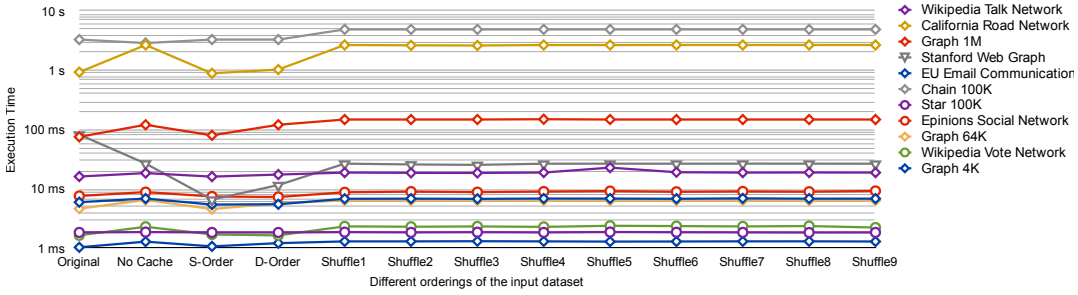


(i) Graph 64K (j) Wikipedia Vote Network (k) Graph 4K

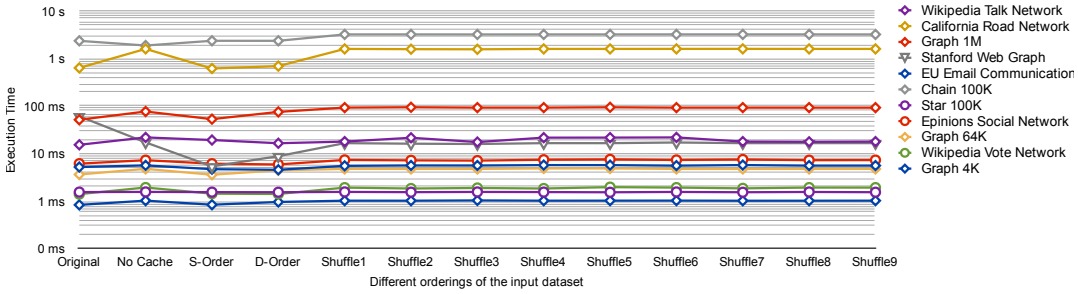
Figure 4.3: Impact on performance for the parallel BFS, when utilizing an increasing portion of the platforms capacity. Evaluated on three different platforms.



(a) Intel Xeon CPU



(b) Nvidia Tesla GPU



(c) Nvidia GTX GPU

Figure 4.4: The Effects the ordering of input edges has on the BFS performance. On the horizontal axis, from left to right, are (i) the original input order, (ii) edges ordered based on the source, (iii) edges ordered based on the destination, (iv) edge lists shuffled nine time using a pseudo-random generator.

caching abilities in the algorithm. This cache unfriendly shuffle scatters the edges from source x over the list of edges, and applies this for each $x \in V$.

Figure 4.4 illustrates the impacts of shuffling on the different systems, with each sub-figure showing the full collection of data sets. The figure shows (almost) no changes in performance for any of the pseudo-random shuffled lists, and only shows a performance improvement for the *Stanford Web Graph* and *California Road Network*, when ordering the edge list according to the source or destination vertex of the edges. So with the exception of these two data sets, the graphs are insensitive to the (re)ordering of the edge list. Thus, applying pre processing on an edge list to gain performance is not worth

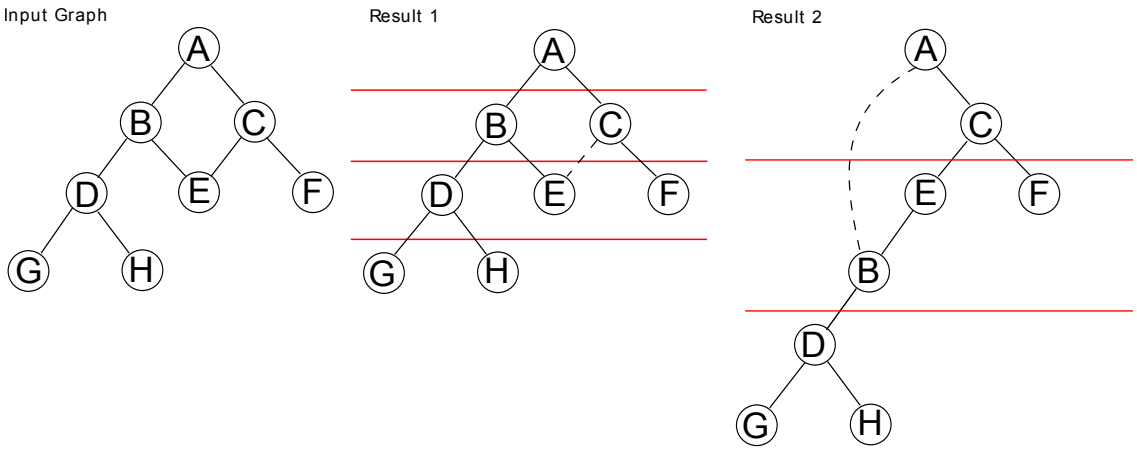


Figure 4.5: Relaxed BFS, a single input graph could result in various search trees with different heights. Where the iterations are separated by the horizontal lines. Note that these differences are either a result of the fluctuations in the timing of threads or of a shuffling of the input edge list.

the effort.

Relaxed BFS

When describing the parallel implementation of our BFS, we encountered a technique of relaxing the BFS rules to gain performance (*Relaxed-BFS*). In this paragraph we validate this assumed improvement by comparison against the parallel BFS implementation. We also show that such an implementation is significantly more sensitive to input ordering. In a BFS algorithm, the performance is bounded by d (i.e. the depth of the resulting search tree), where d is at most equal to the diameter of the graph. Our *Relaxed-BFS*, on the contrary, is bounded by $d' \leq d$. In other words, BFS requires d iterations over the list of edges and *Relaxed-BFS* requires only d' iterations, as illustrated in Figure 4.5, where *Result 1* has $d' = d = 4$ iterations and *Result 2* has $d' = 3 < d$ iterations. As the figure also shows, the resulting search tree could give a false depth perception. However, we are still able to correctly retrieve various graph statistics.

A *Relaxed-BFS* is highly sensitive to input ordering, meaning, a change in the order of edges will lead to a change of d'. In a optimistic case the new d' will approach 1 and in pessimistic case it will reach d.

To show the impact of relaxing the BFS structure, we ran experiments with the same collection of data sets (from Table 2.1). Figure 4.6 illustrates the speed up gained by relaxing the synchronization. The figure shows that, the fine-grained parallelism of GPUs can lead to significant speed up of the *Relaxed-BFS* over BFS. The Intel Xeon shows no speed up, and can even show degradation of performance. We believe this

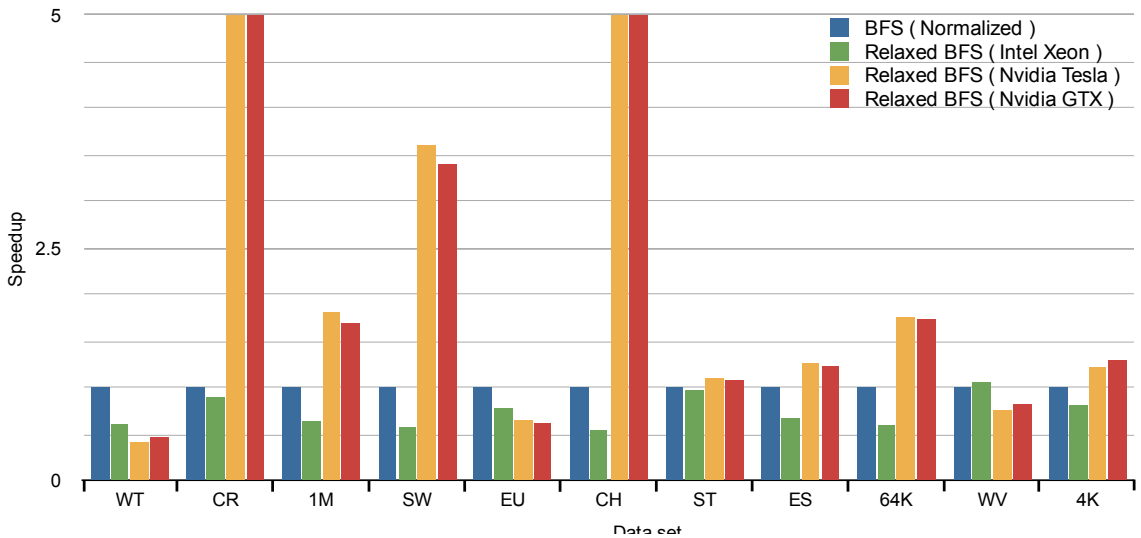
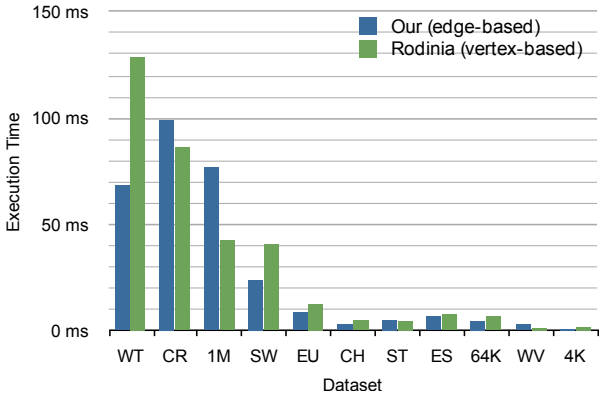


Figure 4.6: Relaxed BFS, a single input graph could result in various search trees with different heights. Where the iterations are separated by the horizontal lines. Note that these differences are either a result of the fluctuations in the timing of threads or of a shuffling of the input edge list.

comes from the way the system is scheduling the fibers and from this get less efficient memory access patterns. This experiment on *Relaxed-BFS* also shows the impacts of level synchronization. We will not continue experimenting with the *Relaxed-BFS*, since our following algorithms (all-pair shortest path and betweenness centrality), require the level-based BFS.

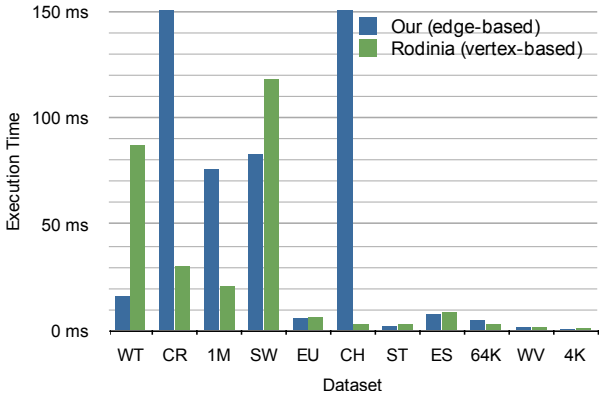
The Rodinia benchmark

As described in section 2.1.2, we use a set of graphs from the repository of Rodinia. This repository is much richer than just collections of data sets, as its main purpose is a benchmark for heterogeneous systems [45]. For our research, this benchmark is a source for evaluating our BFS design: it uses the same programming model (i.e. OpenCL), hence it is applicable to both CPU-based and GPU-accelerated systems, but they make a different choice for graph representation (they use a vertex-based representation). Rather than arguing which representation is best, we can measure the actual behavior of the different representations under the exact same conditions. We use the heterogeneous test environments described in the prior sections – one CPU-based and two GPU-accelerated hardware platforms (Intel Xeon, Nvidia Tesla and Nvidia GeForce, respectively). Although the Rodinia benchmark uses only synthetic data sets, we expand the experiments with real world data sets from the SNAP repository (see Table 2.1 for the full list of data sets).

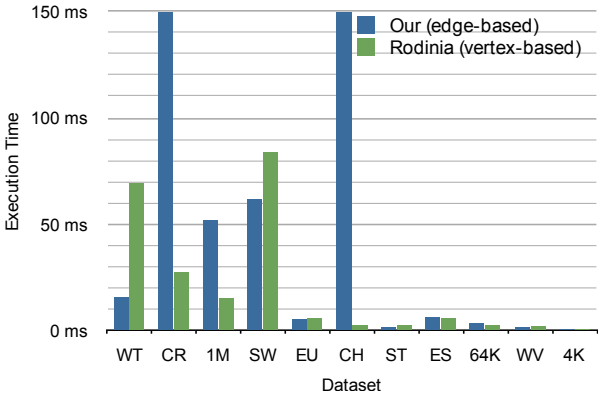


(a) Intel Xeon E5620

	Dataset
WT	Wikipedia Talk Network
CR	California Road Network
1M	Graph 1M
SW	Stanford Web Graph
EU	EU Email Communication Network
CH	Chain 100K
ST	Star 100K
ES	Epinions Social Network
64K	Graph 64K
WV	Wikipedia Vote
4K	Graph 4K



(c) Nvidia Tesla C2050/ C2070



(d) Nvidia GeForce GTX480

Figure 4.7: Comparison of the performance of BFS, for an edge-based versus a vertex-based graph representation: Our implementation, and the implementation from the Rodinia benchmark, respectively.

Figure 4.7 shows the results of our comparison, in terms of execution time, for the various data sets. Note that for the sake of clarity, the data sets are labeled using the abbreviations from the Table 2.1. From Figure 4.7 can be concluded that the choice of a representation will not automatically result in a better performance for any data set. In other words, there is a correlation between the structure of a data set, the data representation and the performance.

Chapter 5

All-Pair Shortest Path

Graph traversals can have various shapes and sizes. Depending on the goal, a specific algorithm can be picked to achieve this. In this chapter, we consider the problem of finding the minimal distance between all pairs of nodes in a graph, called All-Pair Shortest Path(APSP). Finding all shortest paths is a time consuming analysis. The APSP algorithm gets a graph $G = (V, E)$ and finds, for each pair of vertices $u, v \in V$, the shortest path from u to v. The length of a path is the sum of its constituent edges.

An APSP algorithm can be used to determine the diameter of a graph, an useful invariant, which can be used to predict the behavior of an application. The diameter is the maximum number of nodes that need to be passed when going from a vertex to any other vertex in that graph, excluding any path that contains backtracks, takes de-tours or has loops. In other words, the maximum of the found minimal distances between each pair of nodes, is equivalent to the diameter of the given graph. Another application could be the creation of a road atlas: in a graph that represents a road network (where the different nodes represent cities), finding the minimum distances between cities would lead to a clear indication of the travel time.

Johnson's APSP algorithm [23] is an example of a widely used APSP. This algorithm assumes the graph to have directed edges and no cycles. By using the Bellman-Form algorithm [5, 13], all negative weighted edges are converted to become positive, allowing Dijkstra's algorithm [37, 50] to be used for finding the shortest paths. A different approach of solving the APSP, is using a dynamic programming methodology instead of single source algorithms. This methodology tends to solve the problem by decomposing it in independent subproblems, that are much simpler. A widely known of such an implementation, is the Floyd-Warshall algorithm and was developed in 1962 by Robert Floyd [12]. This algorithm compares the possible paths in the graph to find the shortest between any pair of vertices.

All the above mentioned techniques, however require a specific graph representation (i.e. vertex-based adjacency list or matrix). In Chapter 2.1, the discussion on an useful graph representation concluded in a choice of edge-based input. Hence, for an implemen-

tation of APSP based on Floyd-Warshall or Johnson's algorithm we require significant changes in the algorithm. Instead, we want to reuse our prior study of implementing the BFS algorithm. By decomposing the algorithm into a set of subproblems, we can conveniently separate the APSP problem in a collection of single source shortest path problems (i.e. we can use our BFS implementation).

5.1 All-Pair Shortest Path using Breadth First Search

Our previous research on a BFS traversal focussed on finding a shortest path from a source vertex to any connected vertex. Hence, a BFS can provide the minimal distance information from a single node to any other node in the graph. This information can be used as a first step towards the APSP, because an APSP problem is the collection of V shortest path problems, where V is the number of vertices in the graph. By systematically performing a BFS traversal, using different source vertices, the combined results will be equivalent to an APSP.

We can use the implementation of our Edge-Based BFS (described in detail in Chapter

Algorithm 5 An All-Pair Shortest Path implementation, by using a BFS to gather minimal distance for any pair of vertices. With D be the resulting collection of distances between pairs of vertices.

```
1:  function APSP_USING_BFS(V)
2:      D ← ∅
3:      for v ∈ V do
4:          BFS(V, v)
5:          for u ∈ V do
6:              if v ≠ u then
7:                  Add(D, Distance(v, u))
8:              end if
9:          end for
10:     end for
11: end function
```

4), by collecting all depths in the search tree, which correspond to the minimal distance to the source vertex s. Iterating over the list of vertices in the graph and assigning each of these vertices as source of the BFS, will be lead to a solution to the APSP problem. Algorithm 5 describes this process of reusing the BFS for an APSP. Due to the many invocations of the BFS algorithm, small performance improvements in this BFS, lead to significant improvements for the APSP. Furthermore, this *many-BFS* solution for APSP increases the bulk parallelism of the application, making it a better match for massively

parallel architectures. For instance, if the performance of a single BFS is improved with only one second on a graph with a million vertices, the performance of the entire APSP will be 11 days faster. Recall that the BFS algorithm is used to find different kinds of heuristics of a given graph (e.g. in- and out-ratio of edges, distance to a source vertex, the size of connected components). In the APSP, we are only interested in the distances from the source vertex. Hence, stripping the BFS algorithm from any unused invariant will reduce memory use and, more importantly, is likely to gain speed.

5.2 Parallel All-Pair Shortest Path

To accelerate the APSP algorithm, we apply parallelism by using multiple threads. Making use of multiple threads requires splitting the complete task of APSP into smaller (preferably equally sized and/ or self contained) sub-tasks. The core part of an APSP algorithm is finding the shortest path between a pair of vertices, and in our algorithmic choice this means performing a BFS traversal. By using the implemented *Parallel-BFS* suggested in Chapter 4, we have a first parallel APSP.

Although applying a parallel version of BFS should lead to good performance. But, as shown in Chapter 4, there is a turning point where adding more threads has little to no impact on performance. In other words, this APSP approach does not exhibit any increase in parallelism. In fact, the parallelism remains limited to the parallelism of a single BFS, which in turn is highly dependent on the input data set (in terms of size, density, and diameter).

There is another possible approach of parallelizing the APSP: a thread i can perform a complete BFS traversal (with source s_i), while the next thread concurrently executes a BFS with source s_{i+1}. This way each thread will have its own BFS assigned and with T threads, the algorithm can execute T BFS traversals concurrently. Note that not all BFS traversals require the exact same execution time, so in this case the time slot we mention is that of the longest traversal.

Such an approach could theoretically result in the entire APSP to finish in the time of a single sequential BFS. Thus, assigning a BFS to each individual thread is an interesting option because it will allow high level of parallelism for APSP (especially for large graphs). In practice, there are hardware that make such an implementation infeasible. For example, the Nvidia GeForce GTX480 (used for the evaluation) supports 15 groups of work-items, where each group can contain 1024 work-items, leading to a total of over $15,000$ work-items. But the hardware only has a capacity of $3GB$ of memory, which means $0.2MB$ per thread. Is this going to be enough for large graphs? We can estimate this by simplifying BFS, and only consider coloring (i.e. one byte is required to distinguish undiscovered and discovered vertices). With our given memory per thread, we can store 0.2 million colored vertices. This already excludes half of our test graphs.

And we have not even bothered storing any input edge list.

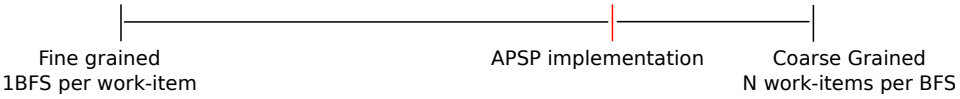


Figure 5.1: Range of parallel implementations of the APSP. Going from fine to coarse grained parallelization. The marker indicates our implementation choice.

Both solutions we proposed so far where unfeasible due to either too low parallelism or too high resource requirements. In Figure 5.1, we show these two extreme cases as the boundaries of a range of possible implementations, with the coarse grained, high latency approach on the far right side and the fine grained, high throughput on the far left side. We choose an implementation that combines both to get a better parallel implementations, feasible for our test systems.

The OpenCL architecture has two distinct layers of parallelism: (i) local and (ii) global. Local parallelization uses an arbitrary number of work-items in a single work-group that contribute on the task at hand. Each work-item is able to have direct communication with any other work-item, and this can be used for synchronization of the system. Global parallelization tends to combine an arbitrary number of local parallelizations, called work-groups, to contribute in the task. Communication between work-groups is not directly supported. This communication limitation requires additional tricks to enforce synchronization (see Section 4.2).
For the APSP algorithm, we can use these two layers of parallelism to gain speedup by applying local parallelization to each BFS, and global parallelization (i.e., multiple work-groups) to concurrently perform different BFS traversals. This design choice will significantly reduce the complexity of BFS synchronization since each group has internal communication possibilities. Given the large number of work-groups supported by the hardware (14 is the minimum for our test platforms), this approach will increase the amount of parallelism in our solution. Furthermore, resource-wise, we can afford to allocate around 200 MB per BFS traversal, which is enough for all our test graphs.

After the assignment of work-items and a choice for the work division, the next step is to orchestrate the APSP by combining the different traversals to construct a result. In the choice of scheduling the different traversals, there could be two types of setups. (i) In *Host-based scheduling*, the host CPU will send a traversal task to each group, this task is executed and the results are fed back to the host. The host will process the results and assign a next traversal to this group. Ultimately, the host collects all results from the different groups and combines them into a final result. (ii) In *Kernel-based scheduling*, the host CPU will assign the entire APSP task to the device. Now each work-group will pick a different task and will place resulting values on a dedicated location in memory. After finishing a traversal task, the group will continue

to a next. Kernel-based scheduling requires a mechanism that ensures the execution of each individual task and a switching mechanism that assigns mutual exclusive collections of tasks to the different groups. When using *host-based scheduling*, such a switching mechanism is not needed. The host can just assign a traversal task to a free group, in a sequential fashion. When a group finishes its task, the host assigns the next BFS to it and this way ensures the completeness of the results. However, this requires additional communication between host and device, plus the device will remain idle when awaiting a new task assignment from the host, with large negative effects on performance. We choose therefore for *kernel-based scheduling*, as we estimate the switching mechanism will be less of a problem than the repeated communication between the host and device. For the scheduling we use a simple iteration over the collection of traversals that, by using G sized steps, and uses an unique identifier per work-group to select a specific traversal task (G is the number of groups used to perform the APSP algorithm). Of course, this simple scheduling method can lead to lower utilization of the platform and/or low performance. As not all BFS trees (i.e. the resulting tree of a BFS traversal) have the same depth, this work distribution might lead to load imbalance: a group i can end up having to execute traversals with only a minimal depth, while group j will have a similar amount of traversals in its queue, but these traversals will all be diameter-deep. In this case, group i will be idle waiting for group j to finish. To solve this imbalance (that leads to low platform utilization), two solutions are possible: (i) use more logical work-groups than physically available cores, resulting in the hardware scheduler striving to give more work to idle hardware (which depends entirely on the platform and its infrastructure), and (ii) design and implement heuristics for data partitioning such that not all long traversals end up in the same work-group (which is difficult because this difference in depth is not known a priori, and might lead to a slow scheduler). Both these options are interesting ideas for future work.

For the implementation of our APSP algorithm, as described in Algorithm 6, we use the BFS design from Chapter 4.2 and make some minor modifications. First, all invariant related operations are stripped off, to have a BFS that solely consist of a coloring and depth computation block. Note that the coloring is always required to ensure the correctness of the output. Next, we introduce the scheduler, an iterative *'for'* statement that selects the root nodes of the BFS traversals at hand (see lines 2, 3 and 20 of Algorithm 6). Note that the root node is used to place the depth values in the resulting distance matrix. This matrix D will collect the distances between any pair of nodes in the graph.

5.3 Experiments and results

For the evaluation of our APSP algorithm, we compare the OpenCL implementations on both CPU-based and GPU-accelerated systems against a sequential implementation.

Algorithm 6 Parallel APSP implementation in OpenCL, for graph $G = (V, E)$. Where D is the resulting distance matrix and $E_{thread} \subseteq E$ the edge list assigned to this work-group.

```
1:  function KERNEL_APSP(E, V)
2:      r ← getGroup()
3:      for r ∈ V do
4:          Q_i ← {V_s}
5:          hasNextLevel ← 1
6:          i ← 0                                          ▷ Set Current Level
7:          while hasNextLevel = 1 do                      ▷ Begin BFS
8:              for e ∈ E_thread do
9:                  if e = UNVISITED & getSource(e) ∈ Q_i then
10:                     add(Q_{i+1}, getDestination(e))
11:                     e ← VISITED
12:                     D[r][getDestination(e)] ← depth[getSource(e)] + 1
13:                     atom_xchg(hasNextLevel, 1)
14:                 end if
15:             end for
16:             i ← i + 1                                  ▷ Increase Level
17:             barrier()                                  ▷ Level Synchronization
18:         end while                                      ▷ End BFS
19:         barrier()
20:         r ← r + getGroupCount()
21:     end for
22: end function
```

Given a graph with N nodes, the sequential APSP implementation uses our sequential edge-based BFS, executed in a loop that runs N times (with N different root nodes).

On top of this, we compare the performance of our parallel algorithm against a predicted execution time (*Expected*) and the theoretical performance on a system with enough resources for the fine grained implementation of APSP (i.e. all BFS traversals are performed concurrently), referred to as ∞ *processors*. For the prediction of our APSP, we use the execution time of our prior implemented BFS and multiply it with the number of traversals that would be assigned to each of the G groups (see Equation 5.1).

$$T_{predict} = T_{BFS} \cdot \left\lceil \frac{N}{G} \right\rceil \tag{5.1}$$

We do not expect this simplified prediction model to lead to very accurate predictions, since we assume the time required for calculating the various invariants is equivalent to the overhead of scheduling. But the prediction should give a good indication on the

	1 Work-item per work-group	1K work-items per work-group
WT	3:45:52.722	21:10:34.057
CR	2:57:56.792	23:17:33.464
1M	3:41:06.257	9:16:33.593
SW	0:09:46.029	0:49:27.121
EU	0:02:03.445	0:04:24.596
CH	0:00:15.725	0:00:32.032
ST	0:00:12.797	0:00:26.828
ES	0:00:33.766	0:00:56.097
64K	0:00:23.698	0:00:46.705
WV	0:00:00.613	0:00:00.891
4K	0:00:00.088	0:00:00.249

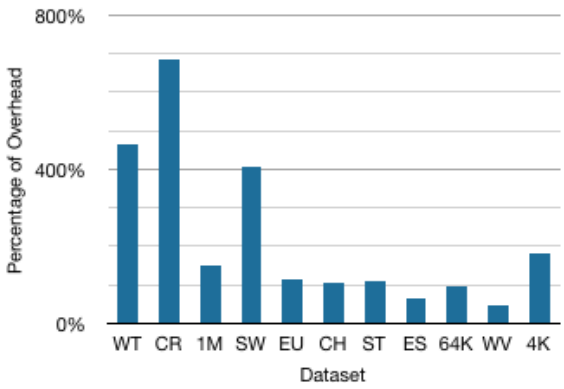


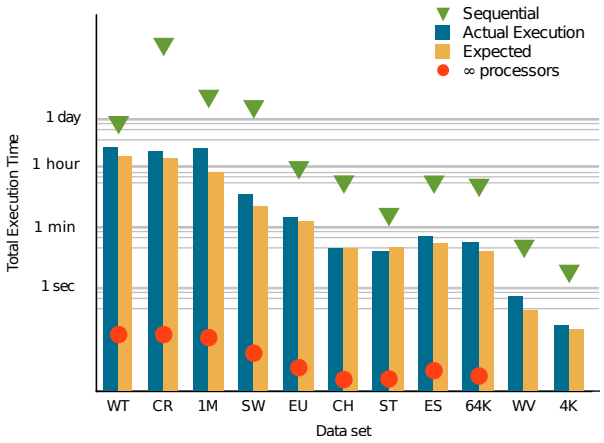
Figure 5.2: As result of the fine-grained OpenCL model, CPU-based systems use fibers that causes mayor impact on performance. By using a single work-item per work-group the performance of CPU-based systems improve.

actual performance gain/loss. For the ∞ *processors*, we estimate the execution time by using the sequential BFS execution time, because the ∞ *processors* implementation reflects the parallel design where we have the capacity of concurrently performing all BFS traversals. Thus, the execution time of the APSP would be equal to the worst case execution of a single sequential BFS (i.e. the BFS traversal where the resulting tree has the diameter as depth).

We use the same experimental setup as the one for BFS, described in Section 4.3.

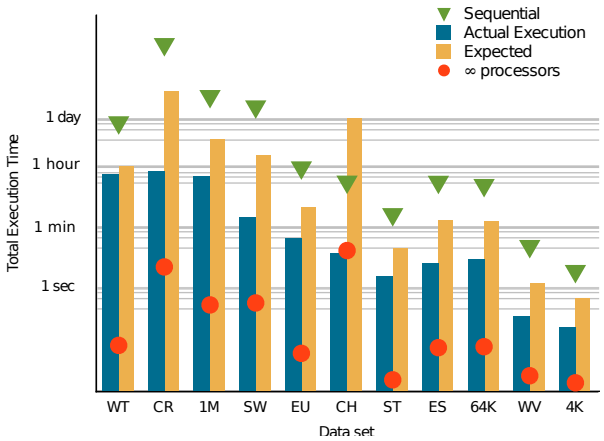
In our initial experiments, the CPU-based system shows unexpected behavior: in all cases the actual performance is significantly worse than our calculated expected performance. For some data sets, the performance of the sequential implementation outperformed our parallel implementation (a behavior which we did not expect to happen for this algorithm). The reason for this behavior was due to the way CPUs emulate work-items by using fibers. As these fibers are sharing resources, when too many of them are actively executing, they will not be able to run concurrently, and will be serialized. In our experiment on BFS (see Section 4.3), the overhead caused by these fibers was close to zero and we assumed it to be negligible. However, as shown in figure 5.2, the overhead of fibers when executing an APSP algorithm becomes significantly large, causing this unexpected behavior. So we altered our experimental setup by changing the number of work-items per work-group to 1 (for the CPU only).

Figure 5.3 shows the performance of our APSP implementation and the above explained reference values (sequential, expected, and ∞ processors execution times), with each sub-figure showing the execution times (of all our test data sets) on a different system. Note the logarithmic scale of the execution time.

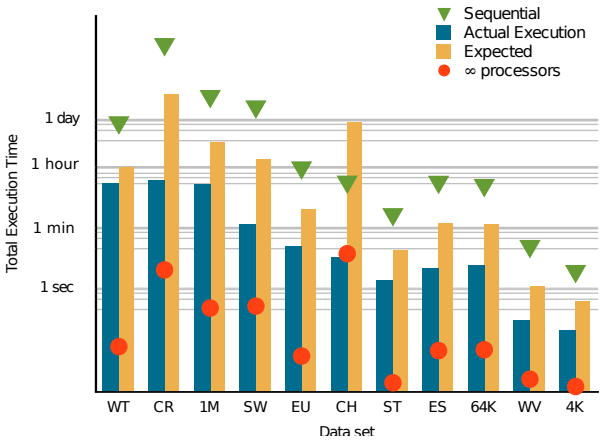


(a) Intel Xeon E5620

	Dataset
WT	Wikipedia Talk Network
CR	California Road Network
1M	Graph 1M
SW	Stanford Web Graph
EU	EU Email Communication Network
CH	Chain 100K
ST	Star 100K
ES	Epinions Social Network
64K	Graph 64K
WV	Wikipedia Vote
4K	Graph 4K



(c) Nvidia Tesla C2050/ C2070



(d) Nvidia GeForce GTX480

Figure 5.3: Performance of All-Pair Shortest Path using OpenCL. The vertical axis shows the execution time in a logarithmic scale.

A significant performance improvement on the GPU accelerated systems (Figure 5.3c and 5.3d) compared with the sequential implementation can be observed, with an almost constant difference between the two GPUs. The GTX performs better, because it has both a higher clock frequency and a higher memory bandwidth than the Tesla. On the other hand, Tesla has more memory and more cores (i.e., streaming multiprocessors), which means it could support higher application parallelism to achieve better performance. Thus, since we aim for a generic implementation for all systems, and one limited by the smallest of these systems, Tesla is slightly disadvantaged. The CPU-based system (Figure 5.3a) also shows significant improvement compared with the sequential version, but both GPUs perform better for all data sets. The achieved speedups are between 1.4 and 11.4 for GTX, and between 1.2 and 6.8 for Tesla (with an average of 6.66 and 4.28,

respectively).

From our experiments, we can conclude that the performance of our APSP on CPUs is as expected: it shows a linear increase when increasing the number of BFS traversals. For the GPU, we note that the performance we achieve is much better than the one expected, because our assumptions when using the worst case scenario for the performance of BFS is too pessimistic. The execution time is also much worse than the one predicted for infinite processors, mainly because the gap between this ideal platform (using N processing units, where N can be as large as a few millions) and the real hardware (with only a few hundreds processing units) is very large. More realistic bounds are required for a real performance prediction. Still, the current predictions are suitable as worst-case scenarios, and show that our implementation is far from these extreme cases.

The behavior of both types of systems, however, is in a sense as expected: recall that in our design choices we used an approach that maximized the granularity while staying within the resource limitations, making the GPUs profit from their finer-grained parallelism and the CPU suffer from the more coarse-grained architecture (i.e. the CPU is limited to parallelism through work-groups). Furthermore, we note that our incremental increase of complexity (i.e., from BFS to APSP) has delivered the expected results as well: the amount of parallelism the problem exhibits has increased significantly, leading to a significant improvement in the performance of the parallel platforms over the sequential ones. Finally, for this APSP implementation, we note that the "more parallel" architectures (i.e., the GPUs) seem to win the performance battle, especially for larger graphs.

Chapter 6

Betweenness Centrality

In the analysis of graphs, algorithms like BFS and APSP are used to retrieve some global statistics of the graph, like diameter or density. Such statistics give a rough overview of the graph, but they show no insight on its actual structure. Centrality analysis is a technique that provides more detailed information of the individual vertices in the graph. With a centrality analysis, we measure the influence a vertex has on the connectivity of the graph. BFS can also be used for a simple centrality analysis, namely *degree centrality*. This analysis measures, in some sense, the popularity of a vertex (i.e. it measures the in- and out-degree of each vertex). Another centrality index, *closeness centrality*, can be found using APSP. Closeness centrality focusses on the distances between all pairs of vertices, hence the length of the shortest paths from any vertex to any other vertex in the graph.

In this chapter we focus on the *betweenness centrality* (BC), a measure of the ratio of shortest paths passing through a vertex. For example, given a graph that represents a road network, the betweenness centrality can provide insight at which junctions/ roads traffic jams are likely to occur.

A widely used BC algorithm, proposed by Freeman [14], searches for all the shortest paths between any two vertices and assigns a degree of betweenness (between 0 and 1) to the intermediate vertices. An intermediate vertex has a degree of 1 if and only if all the shortest paths between two other vertices pass through it, and 0 if no shortest path passes through it. The BC index of a vertex v is the sum of degrees of betweenness for all pairs of vertices (see equation 6.1). Here, σ_{st} denotes the total number of shortest paths between s and t, and $\sigma_{st}(v)$ the count of shortest paths that pass through v.

$$BC(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{6.1}$$

Figure 6.1 illustrates an example graph in which the BC indices are calculated. For simplicity, only a single source vertex is considered in the figure. The image shows that

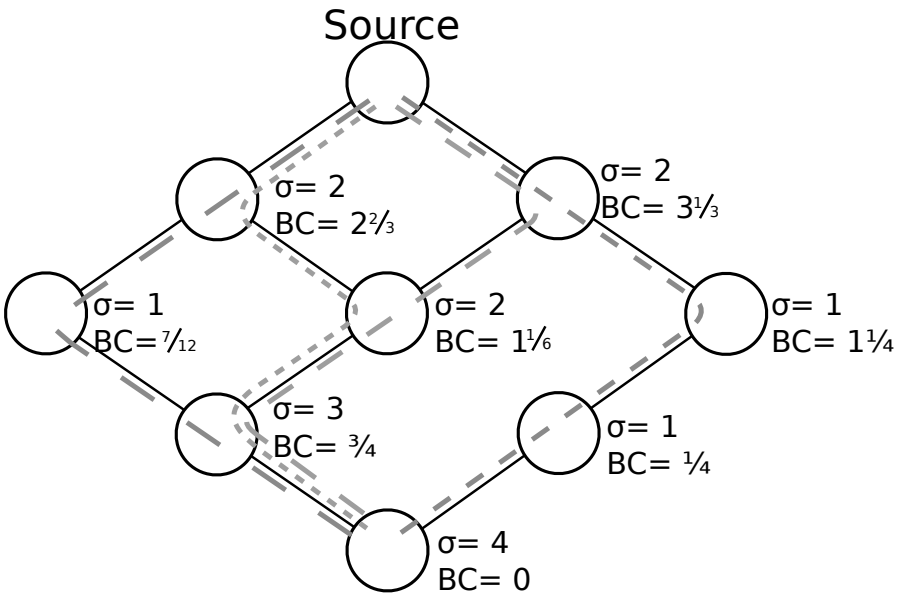


Figure 6.1: Betweenness Centrality, the ratio of shortest path travelling through a vertex on the route from source to each of the other vertices. Note that the values shown are based on a single source vertex to reduce complexity.

the BC indices are based on the paths (i.e. the dotted lines). These paths do not have to travel from source to the bottom most vertex, all intermediate vertices are also valid end points that need to be included in the BC index. For instance: the right most vertex is part of a fourth of the paths from the source to the bottom most vertex, is on all the paths to the vertex beneath it and is on no other path. So the BC index of this vertex (based on the single source) is $1\frac{1}{4}$.

To compute the exact BC score for every vertex, an all pair shortest path problem needs to be solved, which makes this a time consuming task. To reduce time and complexity, simpler indices have been proposed to obtain approximations of the betweenness centrality, like in [11]. Here the algorithm is changed by simplifying the graph into an ego network (i.e., a graph that is originated from the perspective of a single vertex) and using the betweenness centrality based on this simplified perception of the graph. In [7], the focus is on reordering the steps in the betweenness algorithm to reduce the overall time complexity. The time complexity is reduced from $\Theta(n^3)$ to $\mathcal{O}(nm)$ for unweighted graphs. This approach is used and extended by [25] to make it run in a parallel environment.

We choose to extend our existing APSP algorithm to integrate betweenness centrality in OpenCL. To extend the APSP, with limited additional computational time, we apply the techniques proposed in [7] and [25].

6.1 Betweenness Centrality using All-Pair Shortest Path

To calculate the BC of graph, we have to compute the number of shortest paths between pairs of vertices, remember the vertices on each of these paths, and determine the ratio of shortest paths passing through each vertex in the graph. Note that the paths are not considered to pass through the start and end vertices, hence they will always get a 0 *pair-dependency* (as defined in [7]). This procedure is repeated for all pairs of vertices in the graph, with $(s,t) = (t,s)$, and the pair-dependencies are accumulated per vertex.

From this description of BC, we can distinguish a set of three steps: (i) a traversal to get the total number of shortest paths for a pair of vertices, (ii) computing all ratios of (shortest) paths passing each vertex for a pair of vertices, and (iii) adding the derived pair-dependencies (i.e., the ratios of shortest paths) for each vertex to its BC value. This sequence of steps is repeated for computing pair-dependencies of all vertices for the different pairs of vertices. We can implement this repetition using the exact same switching mechanism as for APSP (see Section 5.1): a basic iterator over the different root vertices, that calls a BFS for each root vertex. In the case of BC, we replace the BFS with steps (i), (ii) and (iii) described above.

Even though we replace the BFS, we do not discard it completely. For deriving the total number of shortest paths for a pair of vertices (i.e. the first step), we apply a BFS traversal from a specific source vertex and rather than a measure of the depth in the tree we track the number of (shortest) paths for each vertex. The number of paths to a vertex equals the sum of paths leading to its parent vertices. In contrast to a traditional BFS, this traversal will include all vertices in level i, as parent of a vertex in level $i+1$ if an edge between them exists, resulting in a nontree structure: a *search graph*. Note that each vertex in the search graph forms a pair with the source vertex, hence will provide a pair-dependency that needs to be added to the BC index of each vertex in the graph.

Now we have the total path counts for any vertex pair (s,t), with s being the source vertex and t being any other vertex in the graph. Based on the technique presented in [7], the ratio of these counts can be computed for the intermediate vertices. This problem is solved using a reversed order BFS traversal (i.e., *bottom-up BFS*). For this, we start with the bottom level vertices and look for vertices located in the level above. This repeats until the source vertex of the search graph is found. At each step of the bottom-up BFS, the score $\delta_{s\bullet}(v)$ of vertex v is computed. First, a ratio of the subpath is computed by dividing the path count at vertex v with the path count at the child of v. Second, this ratio is multiplied with the score of the child plus one. The "plus one" is introduced since the child is not only a intermediate vertex for the remainder of the graph, but also an end point of the subgraph. $\delta_{s\bullet}(v)$ is the accumulated score of all individual children of v, see Equation 6.2.

$$\delta_{s\bullet}(v) = \sum_{c:v \in Parent(c)} \frac{\sigma_{sv}}{\sigma_{sc}} \cdot (\delta_{s\bullet} + 1) \tag{6.2}$$

In the final step of BC, we look for the pair-dependency values assigned to the vertices in the previous step and add them all to compute the final BC value of each vertex.

To summarize, our implementation uses an APSP that include $2 \cdot N$ BFS traversals, allowing us to build upon the existing implementation of these kernels.

6.2 Parallel Betweenness Centrality

In the implementation of a parallel BC, we have similar choices as described in Section 5.2. An implementation that performs a pair-dependency computation, concurrently for each source vertex, requires massive amounts of resources. Such an approach is infeasible in our test environment. And an implementation that iteratively executes a concurrent pair-dependency computation for each individual vertex, will suffer from synchronization overhead. We choose to use a level of parallelism similar to that of our APSP implementation: a single work-group is assigned a pair-dependency computation, and the different work-groups concurrently process such a computation for different source vertices. Other than in APSP, the content of the BC computation has three distinctive steps, each depending on its predecessor. Hence, synchronization is required between the different steps. However, because of the BFS-like structure of the intermediate steps (see Section 6.1), their own synchronization provides a natural separation between the steps. Thus, we can re-use the synchronization from the BFS implementations and add no extra synchronization mechanisms.

For computing the total number of shortest paths from a source to all other vertices, we use a BFS structure to traverse the graph. This BFS allows multiple incoming edges per vertex, given that the edge-source is on the level prior to that of the edge-destination. If an edge is found, the path count $\sigma[v]$ of the edge-source is added to that of the edge-destination (as described on lines 9-20 of Algorithm 7). Now the vector $\sigma[\cdot]$ will have the count of shortest paths from the source to every vertex. And while $\sigma[v]$ is the total number of paths to v, it is also the number of paths passing through v on the route to t, if one such path exists. On lines 22-31 of Algorithm 7, we calculate the pair-dependencies $\delta[\cdot]$ by reverse traversing the search graph and computing the ratio of passing shortest paths for the different vertices. Finally, the δ values are added to the BC index for each vertex. To prevent unwanted stalling of work-groups, this functionality is separated from calculating the pair-dependency. A δ value is the sum of pair-dependencies between the source and every other vertex, hence it requires $N - 1$ additions for a graph with N vertices. This means that, although the pair-dependency could directly be added to the BC index when it is computed, it will result in up to N changes of the BC index (by a single work-group), hence major read and write conflicts and stalling of work-items. By splitting the update of the BC indices and the calculation of δ values, only a single change per work-group is made to the BC index of each vertex.

Algorithm 7 Parallel betweenness centrality implementation in OpenCL, for graph $G = (V, E)$. With BC be the resulting vector containing the BC scores of all the vertices in G. Note that r is used to schedule the subtasks to the different groups.

```
1: function KERNEL_BC(E, V)
2:     r ← getGroup()
3:     for r ∈ V do
4:         Q_i ← {V_s}
5:         σ ← ∅                                                    ▷ Path Count
6:         δ ← ∅                                                    ▷ Pair-Dependency
7:         hasNextLevel ← 1
8:         i ← 0                                                    ▷ Set Current Level
9:         while hasNextLevel = 1 do                                ▷ Step II
10:            for e ∈ E_thread do
11:                if e = UNVISITED & getSource(e) ∈ Q_i then
12:                    add(Q_{i+1}, getDestination(e))
13:                    e ← VISITED
14:                    σ[getDestination(e)] ← σ[getDestination(e)] + σ[getSource(e)]
15:                    atom_xchg(hasNextLevel, 1)
16:                end if
17:            end for
18:            i ← i + 1                                            ▷ Increase Level
19:            barrier()                                            ▷ Level Synchronization
20:        end while
21:        i ← i − 2
22:        while i > 1 do                                           ▷ Step III
23:            for e ∈ E_thread do
24:                if getSource(e) ∈ Q_i & getDestination(e) ∈ Q_{i+1} then
25:                    delta ← σ[getSource(e)] / σ[getDestination(e)] · (δ[getDestination(e)] + 1)
26:                    atom_add(δ[getSource(e)], delta)
27:                end if
28:            end for
29:            i ← i − 1
30:            barrier()                                            ▷ Level Synchronization
31:        end while
32:        for v ∈ V do                                             ▷ Step IV
33:            if v ≠ r then
34:                atom_add(BC[v], δ[v])
35:            end if
36:        end for
37:        r ← r + getGroupCount()
38:    end for
39: end function
```

6.3 Experiments and results

For the evaluation of our BC algorithm, we use a similar approach as for the APSP. We compare the OpenCL implementations on both CPU-based and GPU-accelerated systems against a sequential implementation, against a predicted execution time (*Expected*) and the theoretical performance on a system with enough resources for the fine-grained implementation of BC (i.e. computations of all the pair-dependencies are performed concurrently), referred to as ∞ *processors*. For the prediction of our BC, we use the execution time of computing the δ value of a single source vertex and multiply it with the number of traversals that would be assigned to each of the G groups (see Equation 6.3).

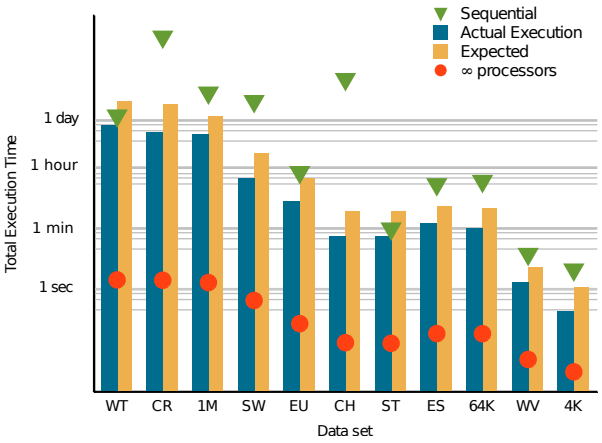
$$T_{predict} = T_{\delta} \cdot \left\lceil \frac{N}{G} \right\rceil \tag{6.3}$$

This simplistic prediction is not very accurate, since we do not include the accumulation of the pair dependencies to construct the eventual betweenness centrality. But the prediction should give a good indication on the actual performance gain or loss. For the ∞ *processors*, we use the execution time of computing the single δ value. Because the ∞ *processors* implementation reflects the parallel design where we have the capacity of concurrently performing pair-dependency calculation of different source vertices. Thus, the execution time of the BC would be equal to the worst case execution of such single δ computation.

We use the same experimental setup as the one for BFS, described in Section 4.3.

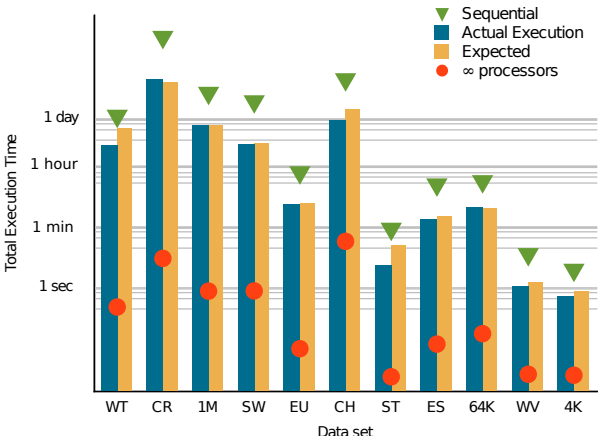
In Figure 6.2, we show the results of our experiments (the measured execution time), as well as the expected, and ∞ processors execution times. Note that the plots present the data sets in a descending order of size, and the execution time in logarithmic scale. Figure 6.2a shows a reduction of execution time on the Intel Xeon for all data sets, compared with the expected performance. Because the time scale is logarithmic, this improvements seem insignificant, but the actual performance is on average 4.6 time better than expected. For the GPUs (Figure 6.2c and 6.2d), the expected value is in most cases a very good prediction of the actual execution time. In some cases the expected time is less than the actual time (e.g., CR, 1M, 64K), other cases (e.g., SW, EU, WV) the actual time is slightly better. Overall, our simple predictor works well for the GPU performance, and can be used as worst-case scenario execution time for the CPU performance.

Finally, we note the performance results in themselves are very interesting. Because the CPU approaches the sequential execution for a few data sets, where other data sets show significant improvements. This variation shows that different graph structures can be more or less CPU-friendly when it comes to the BC computation. On the other hand, GPUs outperform the sequential execution for all graphs, though the gap is not constant (thus, there is some dependency on the graph structure here as well). We believe this

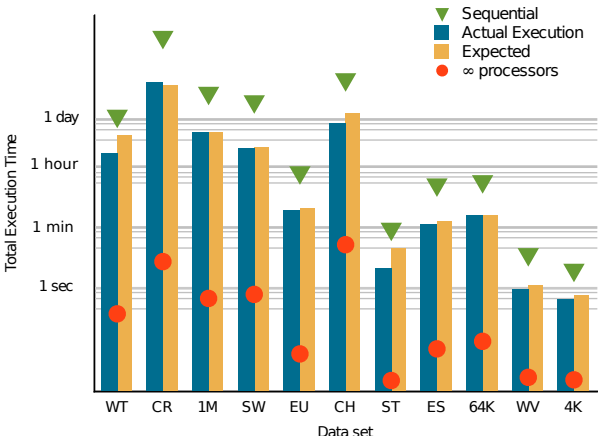


	Dataset
WT	Wikipedia Talk Network
CR	California Road Network
1M	Graph 1M
SW	Stanford Web Graph
EU	EU Email Communication Network
CH	Chain 100K
ST	Star 100K
ES	Epinions Social Network
64K	Graph 64K
WV	Wikipedia Vote
4K	Graph 4K

(a) Intel Xeon E5620



(c) Nvidia Tesla C2050/ C2070



(d) Nvidia GeForce GTX480

Figure 6.2: Performance of Betweenness Centrality using OpenCL.

behavior on GPUs shows again that by increasing the parallelism and/ or complexity of the computation, GPUs can become very important in accelerating large scale graph analysis.

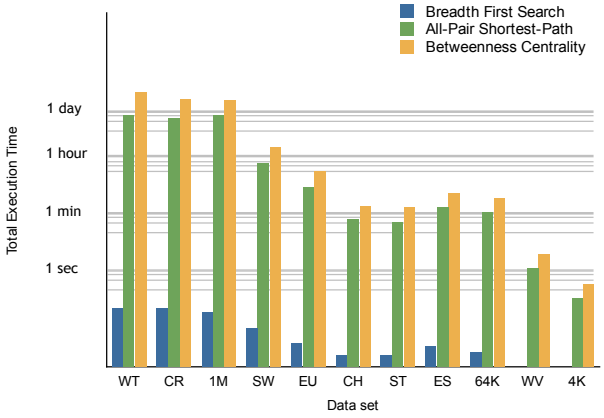
Chapter 7

Discussion

In this thesis we chose our three applications following an incremental complexity approach. We start with the design of BFS, which is a structured traversal of a graph, and we choose a parallel approach that takes into account the edge-based representation for the graphs. However, we show that the level of the parallelism is strictly limited by the graph itself (see Chapter 4 for more detail on both the algorithm and the results). For the design of APSP, we reuse our BFS implementation to traverse the graph starting from each vertex, hence it shows more parallelism than a single BFS (see Chapter 5). Because our APSP consists of multiple BFS traversals, we expect our implementation to reflect both the performance of the parallel BFS, but also the increased parallelism the application exhibits. In other words, APSP is a combination of concurrent executions of the BFS, with each BFS being also parallelized. For our BC algorithm (see Chapter 6), we reuse the APSP design to compute the shortest path in the graph, and add computations to calculate the centrality values. This reuse of APSP suggests the performance of BC to be relative to the APSP, hence relative to the BFS.

In Chapters 4, 5, and 6, we have analyzed each application in isolation, showing its preferences for one platform or another, and for one data set or another. In this chapter, we present two different ways to look at performance. First, we analyze all three applications on each platform, aiming to understand what is the impact of our incremental complexity choice for applications. Next, we also present our results using edges per second, a differently normalized metric used for most graph processing applications as an equivalent to the FLOPs in HPC [25, 51]. Finally, we also include a discussion on the sizes of supported data sets and future solutions to improve the scale of graphs our analyzes can be run for.

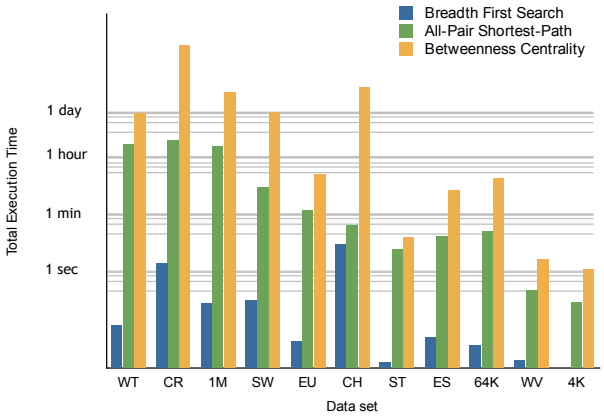
Figure 7.1 shows a comparison between the execution times of each application, grouped per data set and platform. Figure 7.1a represents the performance of a parallel CPU platform. These results show that the increase in complexity leads to an increase in execution time for the parallel CPU platform. Not surprisingly, the APSP takes a lot longer than the BFS, given the scale of the graphs and the limited core-level parallelism



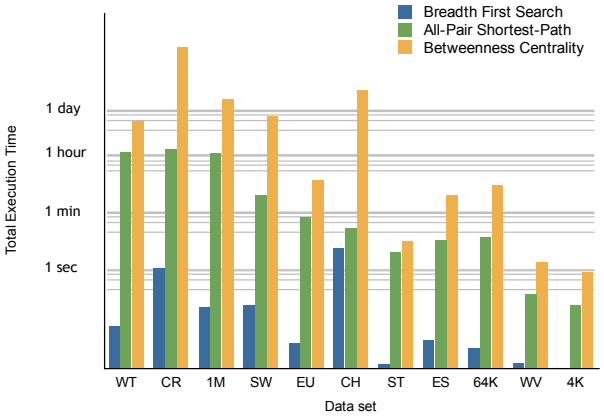
(a) Intel Xeon E5620

	Dataset
WT	Wikipedia Talk Network
CR	California Road Network
1M	Graph 1M
SW	Stanford Web Graph
EU	EU Email Communication Network
CH	Chain 100K
ST	Star 100K
ES	Epinions Social Network
64K	Graph 64K
WV	Wikipedia Vote
4K	Graph 4K

(b)



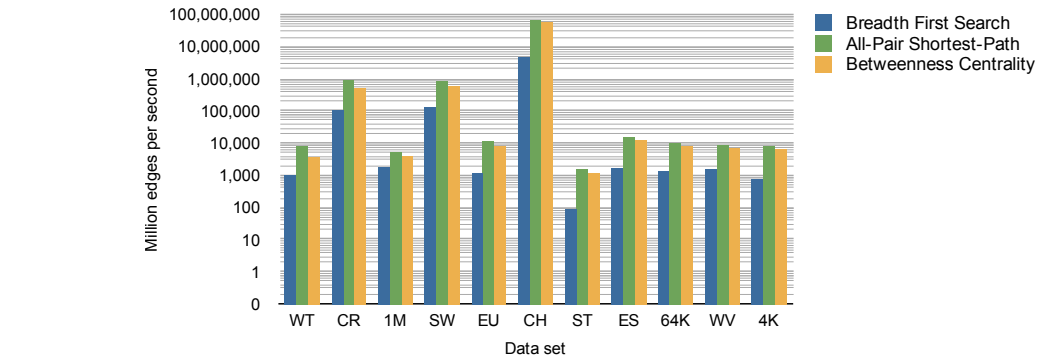
(c) Nvidia Tesla C2050/ C2070



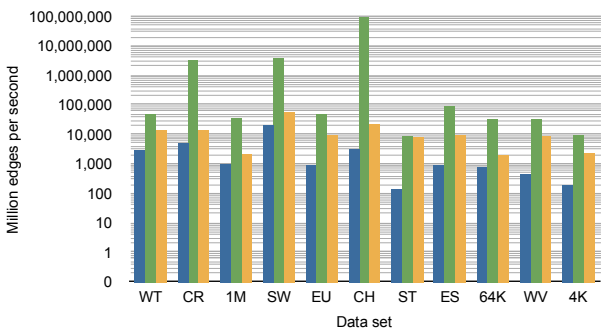
(d) Nvidia GeForce GTX480

Figure 7.1: Performance of our three graph analysis applications. Note that the execution time are shown in a logarithmic scale.

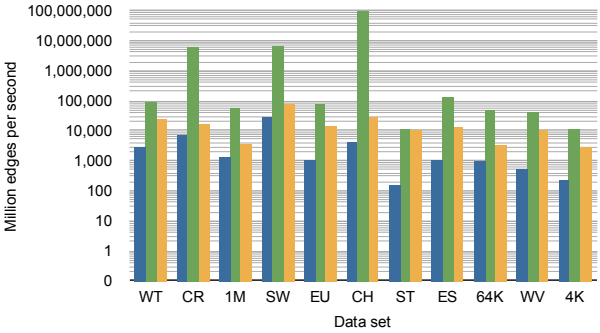
that the CPU offers (a total of 16 parallel threads). Furthermore, the BC shows a doubling of the execution time as compared with APSP – as expected, given the double reuse of APSP inside the BC algorithm. Note that the differences in performance are not the same between data sets, since they differ in sizes and structure. For the Nvidia Tesla C2050 (Figure 7.1c), a large fluctuation can be observed in the performance differences of the three applications. For example, for the *CR* data set, the fraction of the increase in the execution time between BFS and APSP is similar to that between APSP and BC. However, for the *WT* data set, the difference between BFS and APSP is much large than that between APSP and BC. These results show, once again, that the GPUs are much more sensitive to the structure of the data set. The other GPU platform we use, Nvidia GeForce GTX480 (Figure 7.1d), behaves similarly with the Nvidia Tesla, while showing a slightly improved performance for all algorithms (due to more processing power, see



(a) Intel Xeon E5620



(b) Nvidia Tesla C2050/ C2070



(c) Nvidia GeForce GTX480

Figure 7.2: Million edges per second for the BFS, APSP, and BC algorithms, grouped per data set. Note that the execution time are shown in a logarithmic scale.

Table 4.1). Finally, we note that the GPU performance gap between the APSP and BC is in general very large. We believe this is an effect of the dependencies in data for the BC values calculations: the ratios of each of the shortest paths needs to be derived and accumulated for all vertices (i.e., each ratio depend on the ratios of the neighboring vertices, see Section 6.2).

7.1 Another View on Performance

Along this thesis we have used execution time as the main performance metric for our applications. In this section, we offer a different perspective on performance, using the *edges per second* (EPS) metric defined by Graph500[1]. This metric is a normalized view of performance, as it implicitly takes into account the size and the structure of the graph: the size is taken into account in the formula itself, while the effects of the structure are visible in the execution time (as also seen in Figure 7.1, for example).

[1] http://www.graph500.org/

An accurate measurement of the number of edges traversed in total would require the addition of several counters inside the algorithms, which might in turn change the algorithm behavior. Therefore, we choose to estimate the EPS for each of the algorithm by using the theoretical number of edges they will traverse (i.e., based on the algorithm itself).
For our BFS, we multiply the diameter of the graph (D) with the number of edges (M) and divide this with the execution time of the BFS (T_{BFS}):

$$EPS = \frac{D \cdot M}{T_{BFS}} \tag{7.1}$$

For our APSP, we use multiple BFS searches, hence we can use a similar approach for computing the EPS: we multiply the diameter of the graph (D) with the number of edges (M) and with the number of shortest path searches (N), this is then divided by the execution time of the APSP (T_{APSP}):

$$EPS = \frac{D \cdot M \cdot N}{T_{APSP}} \tag{7.2}$$

For our BC, the metric is more difficult to calculate (i.e. it uses an APSP and additional computations to derive the centrality values). In section 6.1, we see that the computations for the centrality values require backtracking of the shortest paths, making it similar to traversing the search tree, in terms of traversal steps. Hence, for simplicity, we can reduce this to an APSP to find the shortest paths in the graph and an APSP to backtrack these shortest path in deriving the centrality values. This simplification allows us to derive a formula for the eps of BC: we multiply the diameter of the graph (D) with the number of edges (M) and with the number of shortest path searches (N), this is divided by the execution time of the BC (T_{BC}). This value is then multiplied by two, resulting in:

$$EPS = 2 \cdot \frac{D \cdot M \cdot N}{T_{BC}} \tag{7.3}$$

In Figure 7.2, the million EPS are shown for the data sets, where each sub-figure represent a different platform. The overall winner in EPS is the APSP algorithm, regardless of the platform or data set. This is because the algorithm we have chosen for APSP is a massively parallel one, and, in combination with the edge-based representation of the graph, it is suitable for the chosen parallel platforms. In the case of BC, the number of edges traversed/ processed per second is lower, due to the additional complexity of the computation performed to determine the BC coefficients. For the GPUs, a very large gap exist between the performance in EPS of the APSP and the other two algorithms. We believe this gap is caused by the large potential of parallelism of our APSP design (i.e., G parallel BFS traversals can run concurrently, where G is the number of work-groups, see Section 5.2), for the CPU however, this is significantly smaller due to

Platform	Algorithm	3 Neighbors	12 Neighbors	24 Neighbors
Intel Xeon (24 GB)				
	BFS	858,993,459	306,783,378	165,191,050
	APSP	169,538,183	58,567,736	31,274,034
	BC	97,612,893	46,684,427	27,531,842
Nvidia Tesla (3 GB)				
	BFS	107,374,182	38,347,922	20,648,881
	APSP	21,192,273	7,320,967	3,909,254
	BC	12,201,612	5,835,553	3,441,480
Nvidia GeForce (1.5 GB)				
	BFS	53,687,091	19,173,961	10,324,441
	APSP	10,596,136	3,660,483	1,954,627
	BC	6,100,806	2,917,777	1,720,740

Table 7.1: The capacity for our three different platform, in terms of maximum graph sizes per algorithm. We present three types of graphs, with increasing average neighbor counts.

coarser level of parallelism in the system. An exception to this gap phenomenon, is the BC algorithm running on the *ST* data set. We believe this is caused by the structure of the *ST*, that allows a fine-grained parallelism also for the BC algorithm.

It is not entirely clear to us why EPS is such a commonly used metric for graph processing. In our experience, it does indeed show a normalized version of performance, but we believe it is still not indicative enough on the impact of the data set structure. The only real advantage one can see here is to compare different implementations of the same algorithm for the same data sets. However, this is not very useful for our case, because EPS does not include any notion of the platform.

7.2 Data Sets and Memory Use

In this thesis, the sizes of our data sets are limited to a few million nodes with a few million edges. With this limitation only small changes where required for our designs to be feasible on GPUs. Larger graphs, however, require more changes in the designs or simply do not fit in the memory of the device. To estimate the boundaries of our design, with respect to data set size, we compute the memory space required to store the (intermediate) information of an algorithm. For completeness, we use three graph structures, with different numbers of average neighbors per vertex (i.e., 3, 12, 24 average neighbors per vertex).

For our BFS, we store the list of edges and the list of vertices, where each edge has an identifier for its source and target vertex, and a color variable (i.e. if the edge is

visited in the traversal or not yet). Each vertex has a depth, in-degree, and out-degree variable. Note that we use the depth for the queuing of vertices (in Algorithm 3) to save space, since it can indicate at what depth it is placed on the queue, hence if it is already queued. To map the variables into memory, we use 4 bytes (i.e. an integer value) for each variable. Thus, our BFS requires 12 bytes for each edge and 12 bytes for each vertex. Our APSP executes G BFS traverses concurrently (where G is the work-group count), each requiring a list to keep track of the queues and a list for coloring. But the BFS used do no need in- and out-degree variables. Thus, our APSP algorithm requires, for 14 work-groups: 64 bytes per edge (one source, one target, and a color for each work-group) and 56 bytes per vertex (a depth variable per work-group). For our BC implementation, we also require a list for the vertex depths at each work-group, but we also need additional variables for counting the number of shortest paths passing through that vertex and for the centrality values (see Section 6.2). Also a separate list is required for the resulting BC values. Thus, our BC algorithm requires 64 bytes per edge (same as APSP), and 168 bytes per vertex (a list for depth, path count, and centrality values for each work-group).
The resulting capacities of our platforms (see Table 4.1) are listed in Table 7.1.

According to Graph500, these are graphs in the scale 21 (i.e., the data set has 2^{21} vertices) or less, which makes our approach suitable for small to moderate graphs, but leaves room for improvement for truly large-scale graphs.

7.3 Multiple GPUs

As seen in Section 7.2, one of the limitations of our current approach is the shear size of the graphs it can handle. To address larger graphs, one solution that should be taken into account is to use multiple GPUs. Consequently, we discuss here the difficulties this approach will encounter.

The first important question for such a multi-GPU architecture is on how to divide the input data set (and therefore the computation itself) between GPUs. Current multi-GPU architectures allow no low-penalty communication between these platforms. Therefore, ideally, the data set will have to be processed on different GPUs without any communication among them. In this case, one can think of different ways to distribute data: equal chunks – independent of the data itself –, connection-based – trying to group connected vertices on the same computing node –, or load-based – trying to separate vertices with a high in-/out-degree on different computing nodes. Research needs to be performed for further evaluation of these strategies, but we suspect there will be no straightforward winner: depending on the data set and the application, different strategies might be required.

Furthermore, a second difficult problem is merging the separately computed fractions of the graph: how and where should the results be gathered. Of course, this operation

depends on the algorithm itself. For example, for BFS, this is a non-trivial problem: the merging of two different BFS trees. We suspect that this problem will be easier to solve by the CPU, due to the sparse structures of these trees, and the predicted memory-boundness of the process itself. Furthermore, the complexity of this tree merging might result in new refinements of the data distribution.
At the first glance, our APSP design is completely insensitive to single- or multi-GPU platforms, due to its separate, independent parallelization over each BFS. However, this assumes that all GPUs will have access to the entire data set, which in turn brings us back to the problem of memory capacity. When the graph is distributed, a reduction phase of combining the APSP distances should be performed by the GPUs, and will eventually require inter-GPU communication. In any case, for distributed graphs, the merging of partial APSP analysis is still non-trivial.
Finally, for BC, both the BFS and the APSP discussion remain valid. Furthermore, in this case, multiple reductions might be needed: after the forward APSP, and a second for the final results. Still, the same strategies should be applied, but the overhead will be increased.

To conclude, we believe moving this graph analysis case studies to the multi-GPU problem requires more fundamental research in two directions: optimal data partitioning and merging of partial results. The first step in this direction would be a thorough study of the out-of-memory graph processing algorithms. We believe this is an interesting next step for future research.

Chapter 8

Summary and Conclusions

The goal of this thesis was to compare the performance of parallel and accelerated parallel graph analysis, where acceleration is achieved with the use of graphical cards. In this chapter we summarize the insights we have gathered from this comparison, and draw a set of conclusions. We further sketch several future work directions.

8.1 Summary

For the interpretation and processing of large amounts of data, graphs are used as an effective data structure. But for large-scale graphs, analysis is extremely time-consuming. By applying parallelism, we reduce the execution time of graph analysis. Introducing more parallelism can lead to better performance, but also increases complexity. In this thesis, we focus on two types of parallel systems: a coarse-grained system with powerful processors and a fine-grained system with more simplistic processors. We consider the coarse-grained system to be a more traditional approach to parallelization, that is constructed out of multiple CPUs. The fine-grained system is an accelerated system, which uses a graphical processor (GPU) for massive parallelization. The graphical processor consists of a large set of processors that work in a concurrent fashion. We note that the major differences are related to the number of cores, core capacities, memory model, and overall parallelism. Specifically, CPUs have relative few but powerful processors and a large amount of memory, while the GPUs have a large number of less powerful processors and access to a relative small amount of memory.

We studied three different graph analysis applications to get a rough estimate on the impacts of the different parallel systems on their performance. We followed an incremental complexity iteration process for defining our applications. Thus, we start with a simple application and extend this towards a complex graph analysis. The first analysis is a search algorithm that traverses the graph a single time, while gathering different kinds of invariants.

With this first algorithm (i.e., a BFS [35]), we found a large variation in performance. With a low level of parallelism, the algorithm seem to favor the CPUs. However, as we increase the level of parallelism (introduce more processors), the GPU approaches the performance of the multi-CPU and for special graphs (i.e. star or star-like structures), GPUs can offer a slight advantage.

Our choice for the next graph analysis application shows much more parallelism, as the analysis involves the execution of a large number of BFS operations. All-Pair Shortest Path (APSP) [23] is an application, we implement by simply running our BFS for every vertex. This approach allows large amounts of parallel operations, by running one BFS per thread. However, during the design stage, we encountered serious limitations for the accelerated system: the GPU is very limited in memory. Thus, we changed our design to require less memory, where threads are grouped each running one BFS. Since the granularity of GPUs is much finer than that of CPUs, the groups of threads can be much larger, hence better performance is expected on GPUs (based on the experimentations of our BFS with the increasing level of parallelism). Our experiments shows the performance gain of the more fine-grained parallelism of GPUs compared to the multi-CPU.

Our simple structure of APSP also allows us to build a simple model for predicting the performance boundaries of our implementation. Our performance boundary model shows an accurate performance indication on the multi-CPU and an upper bound performance prediction on the GPUs. The simplicity of this performance boundary makes it very useful in practice, to help choosing between parallel systems.

The third algorithm, betweenness centrality [14], is a more complex computation that aims to establish the importance of vertices in the graph by counting their participation in the shortest paths in the graph. In our third algorithm, we use a similar prediction for the performance boundary. The third algorithm is a more complex algorithm that measure the influence of the individual vertices, on the distances of the shortest in the graph (i.e. betweenness centrality). Our design reuses the implementation of our APSP, and uses additional traversal steps to backtrack the shortest paths to compute the centrality values. Again, the level of parallelism is limited by our smallest GPU. For the betweenness centrality, our performance boundary is similar to that of the APSP: we use the time of a single computation of centrality values and multiply it with the centrality computations a single group of threads will perform. Our predicted boundary gives an upper bound performance for the multi-CPU and an accurate estimation on the performance of GPUs.

On top of our main research on the impact of accelerators to parallel graph analysis, we also conducted a few experiments on the data set structure, and its relation to performance. We use graphs represented by lists of edges. This choice hides any global overview of the graph from the algorithm, but allows the graph to be insensitive to changes and makes the algorithms applicable to dynamic graphs. In this thesis, we

examine the list of edges through a series of tests that apply different ordering mechanisms and find there is little sensitivity to ordering in our algorithms. However, there is some performance gain from ordering the edge list according to its source vertex. But we believe this gain does not compensate the additional time required for preprocessing the edge list. Also, we conducted a comparison of our 'edge-based' approach against an implementation of the same algorithms that uses a vertex-based graph representation (a list of vertices with its adjacent vertices) from the Rodinia benchmark [45]. In this comparison, we saw both performance degradation and performance improvement for our edge-based approach, depending on the data set. So, if we consider the pre-processing of our edge list into a vertex-based representation, we might not only lose time on preprocessing, but also on performance in the execution. Therefore, we believe that graph representation plays an important role in the performance of the analysis, and it should be regarded as a parameter of the data set.

We also performed a performance study based on the histograms of the data sets. In this part of our research, we try to extrapolate the structure of a graph based on the probabilities a vertex has of having a particular number of neighboring vertices. With a theoretical upper bounded graph structure, we try to categorize our experimental data sets based on their expected relative performance. Here we make a distinction between star-like and better balanced graphs, where star-like means that the majority of the vertices have very few neighbors and a hand full of vertices are heavily connected (like a star), and in better balanced graphs each vertex has about the same number of neighbors. We find that these simple approaches are not sufficient to predict performance. An interesting property we did encounter is that GPUs perform better on data sets we labeled to be star-like and we assume this is caused by the finer-grained parallelism of GPUs (compared to a multi-CPU). We believe this preference of GPUs for star-like graphs is as expected, since the fan out in a star is much larger making it more suitable for massive parallelism.

8.2 Conclusions

The use of GPUs as accelerators for graph analysis shows promising performance. However, it is not superior to the multi-CPU for all algorithms, nor for all data sets. There is a large dependency between algorithm, data set, design choices, and the actual performance, making a selection very difficult. In this research, we show a simplistic performance boundary that can help in the selection of a system for a specific task, with only little calculations required. We also conclude that the graph representation impacts performance, but this impact is not necessarily significant and/ or positive, making preprocessing unwanted and unnecessary. The ordering of the data set shows close to no difference, hence is unnecessary as well. Finally, we show that GPUs have a better performance with data that is clustered in a star-like structure, which we attribute to the

large parallelism of the GPU hardware.

8.3 Future Work

One of the main limitation we encountered in our research is the limited capacity of the GPUs. For the more complex algorithms, the limited resources force us to change the design and reduce parallelism. As future research, the use of multiple GPUs seems like the logical next step. This will bring the advantage of larger graphs (more than the 2.4M vertices in this thesis) at the cost of more complex data partitioning.

References

[1] E. Santos-Neto A. Gharaibeh, L. Beltrao Costa and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. *The 21st international conference on Parallel architectures and compilation techniques, PACT'12*, pages 345–354, 2012.

[2] S. Seufert A. Gubichev, S. Bedathur and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. *19th ACM international conference on Information and knowledge management, CIKM'10*, pages 499–508, 2010.

[3] R. van Nieuwpoort A. L. Varbanescu, P. Hijma and H. Bal. Towards an effective unified programming model for many-cores. *13th Workshop on Advances in Parallel and Distributed Computational Models, APDCM'11*, 2011.

[4] B. Hendrickson A. Lumsdaine, D. Gregor and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 2007.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics, vol. 16*, pages 87–90, 1958.

[6] Y. Boykov and M. P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. *Internation Conference on Computer Vision*, 1:105, 2001.

[7] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[8] A. Chan and F. Dehne. Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters and shared memory machines. *International Journal of High Performance Computing Applications 19*, 2005.

[9] S. Puri D. Tarditi and J. Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. *ASPLOS-XII Proceedings of the 12th international conference*, 2006.

[10] A. Harwood E. Sundararajan and K. Ramamohanarao. Lossy bulk synchronous parallel processing model for very large scale grids. *Computing Research Repository, CoRR'06*, 2006.

[11] M. Everetta and S. P. Borgatti. Ego network betweenness. *Social Networks 27*, pages 31–38, 2005.

[12] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, page 345, 1962.

[13] L. R. Ford and D. R. Fulkerson. Flows in networks. *Princeton University Press*, 1962.

[14] L. C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, page 215, 1978.

[15] A. J. C. Bik J. C. Dehnert I. Horn N. Leiser G. Malewicz, M. H. Austern and G. Czajkowski. Pregel: A system for large-scale graph processing. *The 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10*, pages 135–146, 2010.

[16] S. Gabriel. Altera previews productivity benefits of opencl for fpgas through early access program. *http://www.altera.com/corporate/news_room/releases/2012/products/nr-opencl-eap.html*, 2012.

[17] D. Gregor and A. Lumsdaine. The parallel bgl - a generic library for distributed graph computations. *Parallel/High-performance Object-Oriented Scientific Computing, POOSC'05*, 2005.

[18] C. T. A. M. de Laat D. H. J. Epema D. Koelma F. J. Seinstra H. E. Bal, A. A. Wolters and J. W. Romein. The distributed asci supercomputer 4. *http://www.cs.vu.nl/das4/*, 2011.

[19] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. *HiPC'07 Proceedings of the 14th international conference*, 2007.

[20] C. H. Huang and J. F. Wang. Breadth-first-based decision algorithm for facial biometrics. *TENCON 2009 IEEE Region 10 Conference*, 2009.

[21] Advanced Micro Devices inc. Amd 'close to metal' technology unleashes the power of stream computing. *http://www.amd.com/us/press-releases/Pages/Press_Release_114147.aspx*, 2006.

[22] A. L. Varbanescu J. Fang and H. Sips. A comprehensive performance comparison of cuda and opencl. *The 40th International Conference on Parallel Processing, ICPP'11*, 2011.

[23] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24 issue 1, 1977.

[24] A. Leist K. A. Hawick and D. P. Playne. Mixing multi-core cpus and gpus for scientific simulation software. *Research Letters in Information and Mathematical Sciences, 2010*, 14:25–77, 2010.

[25] K. Jiang D. A. Bader K. Madduri, D. Ediger and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *The 2009 IEEE International Symposium on Parallel and Distributed Processing, IPDPS'09*, pages 1–8, 2009.

[26] G. J. Katz and J. T. Kider Jr. All-pairs shortest-paths for large graphs on the gpu. *23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55, 2008.

[27] R. Menon L. Dagum. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering, vol. 5*, 1998.

[28] J. G. Siek L. Q. Lee and A. Lumsdaine. The generic graph component library. *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'99*, pages 399–414, 1999.

[29] O. Mencer L. W. Howes, O. Pell and O. Beckmann. Accelerating the development of hardware accelerators. *Workshop on Edge Computing*, 2006.

[30] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers, vol. 10*, 1961.

[31] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Computer Vision and Pattern Recognition, CVPR'10*, pages 2181–2188, 2010.

[32] J. Leskovec. Stanford network analysis platform (snap). *Stanford University*, 2006.

[33] J. Liu and J. Sun. Parallel graph-cuts by adaptive bottom-up merging. *22nd ACM symposium on Parallelism in algorithms and architectures, SPAA'10*, pages 303–314, 2010.

[34] M. Khosravani M. J. Dinneen and A. Probert. Using opencl for implementating simple parallel graph algorithms. *Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'11*, 2011.

[35] E. F. Moore. The shortest path through a maze. *International Symposium on the Theory of Switching*, 1959.

[36] A. Munshi. The opencl specifications, version 1.2. *Khronos OpenCL Working Group*, 2011.

[37] D. Gregor N. Edmonds, A. Breuer and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. *The Ninth Implementation Challenge: The Shortest Path Problem*, 2006.

[38] J. Willcock N. Edmonds and A. Lumsdaine. Design of a large-scale hybrid-parallel graph library. *International Conference on High Performance Computing, Student Research Symposium*, 2010.

[39] M. E. J. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical Review E*, 64, 2001.

[40] J. Pearl. Heuristics: intelligent search strategies for computer problem solving. *Addison-Wesley Longman Publishing Co.*, 1984.

[41] R. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal, vol. 36*, pages 1389–1401, 1957.

[42] U. Bondhugula R. Bordawekar and R. Rao. Can cpus match gpus on performance with productivity?: Experiences with optimizing a flop-intensive application on cpus and gpu. *Report RC25033, IBM T.J.*, 2010.

[43] T. Iizuka A. Asahara R. Tsuchiyama, T. Nakamura and S. Miki. The opencl programming book: Parallel programming for mulitcore cpu and gpu. *Fixstars Corporation*, 2010.

[44] J. Meng D. Tarjan J. W. Sheaffer S. Che, M. Boyer and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68 issue 10, 2008.

[45] J. Meng D. Tarjan J. W. Sheaffer S. H. Lee S. Che, M. Boyer and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *The 2009 IEEE International Symposium on Workload Characterization, IISWC'09*, pages 44–54, 2009.

[46] J. W. Sheaffer K. Skadron S. Che, J. Li and J. Lach. A performance study of general-purpose applications on graphics processors using cuda. *The 2008 Symposium on Application Specific Processors, SASP'08*, pages 101–107, 2008.

[47] T. Oguntebi S. Hong, S. K. Kim and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. *Principles and Practice of Parallel Programming, PPoPP'11*, 2011.

[48] J. Sanders and E. Kandrot. Cuda by example: an introduction to general-purpose gpu programming. *NVIDIA Corporation*, 2011.

[49] A. Sriram and K. Gautham. Evaluating centrality metrics in real-world networks on gpu. *High Performance Computing, HiPC'09 Student Research Symposium 2009*, 2009.

[50] V. Osipov U. Meyer. Design and implementation of a practical i/o-efficient shortest paths algorithm. *10th Workshop on Algorithm Engineering and Experiments, ALENEX'09*, pages 85–96, 2009.

[51] K. Ueno and T. Suzumura. Highly scalable graph search for the graph500 benchmark. *The 21st international symposium on High-Performance Parallel and Distributed Computing, HPDC'12*, pages 149–160, 2012.

[52] L. G. Valiant. A bridging model for multi-core computing. *16th annual European symposium on Algorithms, ESA'08*, pages 13–28, 2010.

[53] O. Visser Y. Guo, S. Shen and A. Iosup. An analysis of online match-based games. *IEEE International Symposium on Audio-Visual Environments and Games, HAVE'12*, 2012.

[54] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP'12*, pages 283–284, 2012.