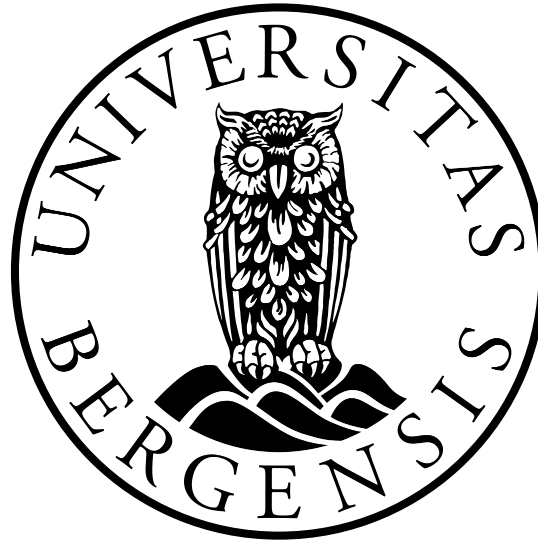
University of Bergen



Department of Informatics

Masters Thesis

K-Core Decomposition with CUDA

Author: Ole Magnus Morken

Supervisor: Fredrik Manne

November 27, 2020

Abstract

In this thesis we create a k-core decomposition of graphs algorithm. We will be using parallel techniques tailored towards a GPU to create our algorithm. These techniques will be employed using Nvidia CUDA, a programming platform that allows us access to the GPU. Our approach to solving the k-core decomposition is based on the locality property, calculating the k-core of each vertex based on its neighbors. We will look at how our algorithm compares to a similar algorithm computed in sequential. Our algorithm will be tested on various graphs, and see how it compared to a similar algorithm in sequential execution. We will discuss our results, and conclude that a parallel algorithm provides a significant speedup versus a sequential one. Lastly, we will discuss what could be done to improve our solution and other approaches such as turning it into an approximation algorithm.

Acknowledgment

Thanks to my supervisor, Fredrik Manne, for guidance.

Thanks to my class mates for all the discussions, and providing me with much need motivation.

Thanks to my parents, who have provided support and love during these hard times.

Ole Magnus M. Morken

Contents

Abstract **ii**

Acknowledgment **iv**

1 Introduction **2**

1.1 Problem statement and motivation 2

1.2 Thesis outline 3

2 Background **4**

2.1 Graphs 4

2.1.1 Definitions, Notation and Terminology 5

2.2 K-cores 5

2.3 Previous Work 6

2.3.1 Smallest-last vertex ordering 7

2.3.2 Cores algorithm 7

2.3.3 Locality Property 8

3 Parallel Computing **10**

3.1 Parallel Systems 10

3.2 Computing in parallel 11

3.3 Graphics Processing Unit 12

4 NVIDIA CUDA **14**

4.1 Programming Model 14

4.1.1 CUDA Programs 14
4.1.2 CUDA Kernel 16
4.1.3 Thread Hierarchy 16
4.1.4 Memory Hierarchy 17
4.2 Libraries 18
4.2.1 Thrust 18
4.2.2 CUB 19

5 K-Core Decomposition 20
5.1 Our algorithm 20
5.1.1 Extended with CUDA 25

6 Results 30
6.1 Hardware Used 30
6.2 Graphs 30
6.3 Timing method 31
6.4 Execution times 31
6.4.1 Total runtimes 31
6.4.2 Kernel runtimes 33
6.5 Discussion 34

7 Conclusion 35
7.1 Further Work 36

List of Figures

2.1 The k-core decomposition of a simple graph 6

3.1 CPU cores vs GPU cores 13

4.1 Thread Hierarchy, Reprinted from[18] 17

4.2 Memory Hierarchy, Reprinted from[18] 18

4.3 Simple example of inclusive sum 19

5.1 Neighborhood of vertex v 22

5.2 Result of make_buckets 22

5.3 Result of the inclusive sum 23

5.4 Possible K-cores of v 24

Chapter 1

Introduction

1.1 Problem statement and motivation

The study of large scale networks and data sets have been interesting for researchers for many decades. These data sets are becoming increasingly large, and doing relevant work is getting more and more taxing on computers. Studying data can be done in many different ways. The key in all of these studies is to find data that is relevant. Many different metrics exist on how to interpret what should be considered relevant. Betweenness, eigenvector, and closeness centrality indices [6] are some of them. *K-core decomposition* is one such established way to analyze graphs.

K-core decomposition has been used to characterize social networks [1], to help visualization of complex graphs [4], to determine the role of proteins in complex proteomic networks [2], and to identify good "spreaders" in epidemiological studies [10].

In this thesis, we will explore how an algorithm for k-core decomposition based on the *locality property* performs in parallel. In graphs, the property of locality is a vertex and its adjacent vertices also known as *neighborhood*. There already exist fast established algorithms for solving the k-core decomposition, but it does not translate very well to a parallel approach. We believe that an algorithm based on the locality property could translate better in a *parallel* setting, even if such an algorithm performs slightly worse in a sequential setting. We will use a computing platform called Nvidia CUDA to design our algorithm for k-core decomposition. Nvidia CUDA is a *heterogeneous* platform that allows us access to computing on the Graphics Processing Unit (GPU).

1.2 Thesis outline

In chapter 2 we provide some background of graphs and k-core decomposition. Furthermore, we mention some previous algorithms designed to solve k-core decomposition. Giving some background of graph theory is necessary as we lead up to our implementation of k-core decomposition.

In chapter 3 we give a brief introduction to parallel computing and how a Graphics Processing Unit functions. We implement our algorithm in parallel, which is why we give a brief introduction to parallel computing.

In chapter 4 we give a brief introduction to Nvidia CUDA. This is the platform we use to create a k-core decomposition algorithm. Nvidia CUDA gives us control over parallel execution on a Graphics Processing Unit.

In chapter 5 we discuss previous algorithms created to solve k-core decomposition in parallel and provide our implementation of the problem both in sequential and parallel.

In chapter 6 we test our algorithm on a selection of graphs and compare how our algorithm performs in parallel versus sequential.

In chapter 7 we conclude why we obtained the results from our tests, and discuss how and why we obtained the results we did.

In chapter 8 we discuss some options to improve our implementation, and discuss other approaches that might suit our algorithm better.

Chapter 2

Background

In this chapter, we give a brief introduction to graphs in graph theory and explain the k-cores of graphs. We provide some terminology that will be used throughout this thesis. Lastly, we present some previous work done solving the k-core decomposition of graphs.

2.1 Graphs

In the field of graph theory, we define a graph as a set of vertices and edges. Vertices are the components of a graph, and edges are the relations between these components. We can picture a social network as a graph. The vertices of such a graph are the users of the social network, and the edges are relationships (such as friends, spouses, colleagues, etc) between them. Depending on what we wish to represent, graphs can take on different properties. We can represent different types of graphs by using various types of edges and by giving attributes to vertices.

Some graphs have *undirected* edges between vertices. An *undirected* edge is a relation that two users share with each other, such as being friends. An edge that is *directed* is a relation that is unique to one of the two vertices who share it. Having a boss is an example of a relation that goes one way. Some relations may carry more importance, or *weight*, than others, and we have what is called a *weighted graph*. We can combine properties to create *directed, weighted graphs* and others. For the k-core decomposition, it makes sense to focus on as simple graphs as possible, as only small adjustments to the algorithm is necessary to make it work on other types of graphs. Therefore, for the purpose of this thesis, we use *undirected, unweighted graphs*, to keep the algorithm uncluttered and simple.

2.1.1 Definitions, Notation and Terminology

We define a graph G as $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We define the number of vertices and edges in a graph G to N and M respectively. The degree of a vertex v is defined as the number of vertices adjacent to v. We denote the degree of vertex v as $deg(v)$, and the minimum degree found in a graph G as $\delta(G)$. Using a social network as an example again, we can picture this as the number of friends a user has. All vertices adjacent to a vertex v are commonly referred to as the *neighborhood* of v. Therefore, adjacent vertices are often referred to as *neighbors*. We denote the *neighbourhood* of vertex v as $N(v)$. A *subgraph* is a smaller graph contained inside a graph, i.e. all of its vertices and edges can be found in the larger graph from which it derives. We denote this as subgraph $H = (V_H, E_H) \subseteq G = (V, E)$ of graph G, where vertices $V_H \subseteq V$ and its edges $E_H \subseteq E$. The notion of the k-core of vertex v is closely related to the degree of vertex v. We will give an introduction to k-cores later in this chapter. K-core of vertex v is denoted as $k(v)$. Furthermore, the maximal k-core of vertex v is known as the *coreness* of v. It is common to refer to vertices as *nodes*, and we will use the two terms interchangeably from here on out. A table of the terminology we will use throughout this thesis is shown in table 2.1.

	Notation
Graph G	G(V, E)
Number of vertices in G	N
Number of edges in G	M
Subgraph H of G	H $\subseteq G$
neighborhood of vertex v	N(v)
degree of vertex v	deg(v)
minimum degree of Graph G	$\delta(G)$
k-core of vertex v	k(v)

Table 2.1: Terminology

2.2 K-cores

The notion of k-core was first described by Seidman [23], although the concept of k-degenerate graphs was formally as early as 1970[12]. The k-degeneracy of graphs closely relates to the concept of k-cores, as it is synonymous with the largest k-core induced on graphs. Formally we introduce k-core as:

- Let G be a graph and H be a subgraph of G.

- Let $\delta(H)$ denote the minimum degree of H.
- Each node in G is adjacent to at least $\delta(H)$ other nodes of H.
- If H is a maximal connected (induced) subgraph of G, with $\delta(H) >= k$, we say that H is a k-core of G.

We can picture k-cores as subgraphs G_k of G, where all vertices in G_k has at least k neighbors also in G_k. A simple example of a k-core decomposition is shown in figure 2.1. Subgraph G_k of graph G, where $k = 1$, is equal to graph G excluding *isolated vertices*(vertices with no edges). K-cores are nested, meaning all nodes in the subgraph G_k, where $\delta(G_k) >= k$, are in the subgraph G_{k-1}, where $\delta(G_{k-1}) >= k-1$. This follows from the definition of k-cores. Vertex u with k-core k has at least k neighbors with k-core larger than or equal to k. Since $k > k-1$, vertex u has more than $k-1$ neighbors with k-core at least $k-1$. For each node, it is thus satisfactory to find the maximal k-core, known as the *coreness* of a node.

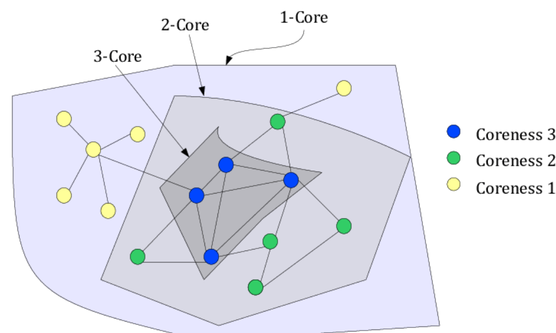


Figure 2.1: The k-core decomposition of a simple graph

2.3 Previous Work

We can find the k-core decomposition of a graph by exhaustively removing nodes with the minimum degree. We start by removing nodes with degree equal to one and decrement adjacent nodes' degree by one. If the adjacent node's degree becomes one after the removal of a node, we remove this node as well and repeat until we have zero nodes with $deg(1)$. When the graph contains no nodes with degree equal to one, we have found the subgraph G_2, where the degree of all nodes is larger than or equal to two. We can now move on to removing nodes with degree equal to two. We execute the same logic on our newly created subgraph G_2 and obtain G_3. We continue to do this until we have removed every node from the graph. By storing each nodes' degree at each intermediate step, we have calculated each nodes' maximal k-core. A simple example of k-core decomposition is shown in figure 2.1.

2.3.1 Smallest-last vertex ordering

An algorithm for finding a smallest-last vertex ordering was created by David Matula and Leland Beck in 1983 [14]. Although this algorithm is not created to find the k-core decomposition, the way smallest-last vertex ordering is computed gives us the k-core decomposition as well. The algorithm is shown in Algorithm 1.

Algorithm 1 Smallest-last vertex ordering

```
1: function SMALLEST_LAST_VERTEX_ORDERING(Graph G = (V, E))
2:     j ← number of vertices in V
3:     H ← G
4:     while j > 0 do
5:         v_j ← δ(H)
6:         H ← H /{v_j}
7:         j ← j - 1
8:     end while
9: end function
```

Finding the minimum degree of G (line 5) is not always trivial, and thus hurt the overall runtime of this algorithm.

2.3.2 Cores algorithm

Another algorithm created specifically to find the k-core decomposition was created by Vladimir Batagelj and Matjaz Zaversnik [3]. The vertices are kept in increasing order based on their degree. After removing a vertex, if an adjacent vertex's degree is updated to be smaller, reorder accordingly. The algorithm is shown in figure 2.

Algorithm 2 Cores algorithm

```
1:  function CORES_ALGORITHM(Graph G = (V, E))
2:      Compute the degrees of vertices in V;
3:      Order the set V in increasing order of degrees;
4:      for all v ∈ V do
5:          for all u ∈ N(v) do
6:              if deg(u) > deg(v) then
7:                  deg(u) ← deg(u) - 1;
8:                  reorder V accordingly;
9:              end if
10:         end for
11:     end for
12: end function
```

Reordering the vertices in V (line 8) is done efficiently with *bin sort*. When the algorithm terminates, the k-core of a vertex v is equal to the value stored in $deg(v)$.

2.3.3 Locality Property

The algorithms described previously in this chapter focus on removing vertices from a graph to compute larger and larger k-cores. In this subsection, we introduce another approach, where each vertex calculates an estimate of its k-core by looking at its adjacent vertices. In the paper Distributed k-Core Decomposition [17], the authors describe an approach based on the *locality property*.

Formally, the locality property is introduced as:

$\forall u \in V : k(u) = k$ if and only if:

1. There exist a subset $V_k \subseteq N(u)$ such that $|V_k| = k$ and $\forall v \in V_k : k(v) >= k$ and
2. There is no subset $V_{k+1} \subseteq N(u)$ such that $|V_{k+1}| = k+1$ and $\forall v \in V_{k+1} : k(v) >= k+1$

The locality property allows us to give an estimate of a vertex's k-core based on its neighbors. After each vertex has computed its estimate, new estimates can be computed based on the previous estimates. This process is repeated until estimates converge, and every vertex estimate corresponds to its true coreness. Since every vertex computes its k-core estimate independently, we can easily structure the computation in *parallel*. Before we introduce our own implementation, we will provide some background in parallel computing. Furthermore,

we will introduce the computing platform the implementation of our k-core decomposition algorithm utilizes.

Chapter 3

Parallel Computing

From 1986 to 2002 the performance of microprocessors increased on average 50% per year [22]. With such a large increase in performance software developers were content with making applications that executed on a single microprocessor, knowing the performance would be substantially better in just a few years. However, this large increase in performance could not last. When the size of microprocessors decreased, their speed could continue to increase. However, this generates a lot of heat and air-cooled integrated circuits are reaching their ability to dissipate this heat [7]. Making processors denser does not generate more heat, so the solution was to fit more integrated circuits on one chip. Adding more microprocessors does not improve the performance of most sequential programs, as it does not affect programs designed for one processor. Computing in parallel, utilizing multiple processors in programs and applications, is required. In this chapter, we give a brief introduction to parallel computing, which is the nature of how GPUs function. Later in this thesis, we explain how we can make programs executing on the GPU, and present some techniques we will use to solve k-core decomposition.

3.1 Parallel Systems

Today most consumer grade computers support parallel execution. However, this was not always the case. To compute in parallel we need systems that support it. Flynn's Taxonomy is a widely used classification scheme to classify different computer architectures[5]. It gives us a general idea of how different systems operate when solving computational tasks. There are four main different schemes a machine or computer may have:

- Single Instruction Stream-Single Data Stream (SISD)
- Single Instruction Stream-Multiple Data Stream (SIMD)

- Multiple Instruction Stream-Single Data Stream (MISD)
- Multiple Instruction Stream-Multiple Data Stream (MIMD)

An *Instruction Stream* is the sequence of instructions performed by a machine, and a *Data Stream* is the sequence of data called for by an instruction stream. Essentially the instruction stream is what work is done by a machine, and the data stream is what the machine does the work on.

SISD is how a standard *single-core* computer is structured. Being *single-core* simply means having one processor. One instruction is executed at a time on one input stream.
SIMD is a parallel structure where multiple identical computations are executed on different data. A multicore computer is structured this way, and CUDA uses a similar scheme.
MISD is an uncommon architecture. Streams are not identical, but they execute on the same data.
MIMD has many streams, that execute independently of each other.

3.2 Computing in parallel

Programming in parallel can be done in different ways. The main idea is to divide a program's workload between processors, utilizing as much computational power as possible. One way is to divide a sequential program into sections, splitting the workload of these sections between available processors. Dividing a program into sections, or *tasks*, and having separate processors execute these tasks is known as *task-parallelism*. Another way is to divide the input data. Distributing the input data between processors, where each processor execute the same task, is known as *data-parallelism*.

When we compute applications in parallel, cores often depend on the execution of other cores. They need to *communicate* between each other. The way cores communicate depends on the underlying system we are making our application in. In a *distributed-memory* system, cores have their private memory. When a core needs the result from another core it has to be transmitted with send and receive functions.

Some systems offer a *shared-memory* structure. As the name suggests, the memory is shared between all cores contained in the system. This eliminates the need for explicit communication, but other hazards accompany instead. When more than one process tries to access the same memory space, they create a *race condition*.

Process A tries to read what Process B has computed and stored in memory space M. In theory, the processes execute simultaneously, and A should have no problem reading the

result from B's computation. In practice, however, the processes do not execute at the same time. If A makes the read-operation before B has stored its result, A reads the wrong value from M. This creates undefined behavior in our program.

To stop *race conditions* from happening, we need to *synchronize* our processes. A synchronization-barrier keeps processes idle while they wait for other processes to finish. It ensures that no work is done until every process reaches the barrier. If we synchronize process A and B after B does the store-operation and before A makes the read-operation, we eliminate the race condition.

3.3 Graphics Processing Unit

The term Graphics Processing Unit (GPU) was first coined by NVIDIA in 1999 [15]. The GPU was initially designed with one task in mind, to offload and accelerate 2D or 3D rendering from the Central Processing Unit (CPU), working through the GPU pipeline[13] to visualize data to a computer screen.

This pipeline has since been optimized, combining several steps in the pipeline together. One key observation paved way for parallelization; Each pixel on a screen can be calculated independently. Consequently, GPUs are designed with a focus on parallel execution.

GPUs focus on high *throughput*, whereas CPUs focus on low *latency*. Throughput is the measure of how many operations can be processed per unit of time. Latency is the amount of time it takes for an operation to start and complete.

The demand for better and more streamlined graphics have accelerated the development of GPUs drastically. GPUs have thus evolved to become powerful computational units, and have been used to accelerate the progress in many fields of computer science.

Nvidia's GPUs are manufactured with various architectures. Some are designed with optimal graphics for gaming in mind, while others focus on more computational power for machine learning and deep learning [19]. All architectures are built around a scalable array of Streaming Multiprocessors (SM). The design of SMs depends on the architecture, but the computational power comes from the specialized CUDA-cores found in all of them.

Today, high-end CPUs have 20-28 cores [9] and consumer-grade CPUs usually cap at 8. Contrary, GPUs available to consumers have thousands of cores. A simple representation of a GPU's cores next to a CPU's cores is shown in figure 3.1.

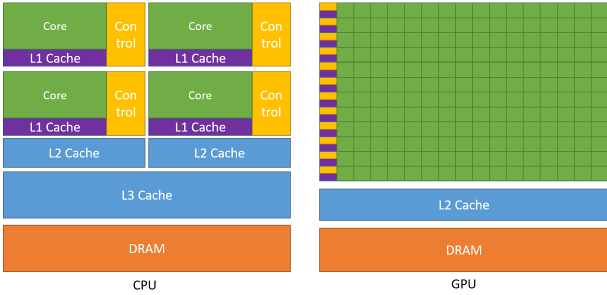


Figure 3.1: CPU cores vs GPU cores

Chapter 4

NVIDIA CUDA

NVIDIA CUDA is a general-purpose parallel computing platform that utilizes NVIDIA GPUs to solve computational problems more efficiently than on a CPU. NVIDIA released CUDA in 2006, and it is currently in its 11th edition. It provides support to many languages such as C/C++, Fortran, and wrappers for Java and Python. For the purposes of this thesis, we will be using C/C++. CUDA is a heterogeneous platform, meaning the GPU and the CPU are both utilized. Even though the GPU offers great computational power, it lacks a lot of the control the CPU offers. We still depend on the CPU to provide relevant data for computation. In this chapter, we give a brief overview of how CUDA functions, and how will be using the platform and some libraries associated with it to solve k-core decomposition.

4.1 Programming Model

CUDA offers higher control over resources than conventional programming. This gives programmers more options when creating programs, but it demands greater care. In conventional sequential programming we typically do not think about how memory stores and loads are done. With CUDA, we are offered explicit control over various levels of memory. Much of the development process of programs utilizing CUDA boils down to managing resources in the form of memory.

4.1.1 CUDA Programs

When we make programs in heterogeneous systems such as CUDA, we need to manage both the CPU side and the GPU side. In CUDA we refer to the CPU to as the *host*, and the GPU as the *device*. These terms are used in most CUDA-functions, and we will use the terms

interchangeably as well. A typical CUDA program is structured as follows:

1. Allocate GPU memory
2. Copy data from CPU memory to GPU memory
3. Invoke a CUDA-kernel to perform computation
4. Copy data back from GPU memory to CPU memory Destroy GPU memory

The GPU and CPU do not share their memory, which is why we need to allocate memory to the GPU and transfer data from the CPU. Since the GPU does not offer input functions, it depends on the CPU to provide useful data needed for computation. A CUDA-kernel is a function that enables GPU computation from the programmer's perspective. We will explain CUDA-kernels in detail later in this chapter. After the computation is finished on the GPU, we transfer the result back to the CPU if needed. Similar to garbage management in conventional C-programs, GPU memory must be freed after we are finished with it. A simple CUDA program is shown in Listing 4.1. The type of `devPtr` is float, but it could be any other datatype supported by CUDA. We assume `hostPtr` is declared earlier in the scope, and that it is of the same type as `devPtr`. The argument `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` is of the CUDA-type `cudaMemcpyKind`, and describes which way we want to transfer data. The `simple_kernel< < <,> > >()`-function (line 9 in Listing 4.1) is the CUDA-kernel that enables work to be done on the GPU which will be explained in detail in section 4.1.2. BLOCKS is the number of *blocks* we wish to invoke the kernel with, and BLOCK_SIZE is the number of *threads* in each block. *Blocks* and *threads* are central terms necessary to understand how CUDA programs work, which we will explain in detail later in this chapter.

Listing 4.1: Snippet of a simple CUDA program

```
int main()
{
    ...
    float *devPtr, *hostPtr;

    cudaMalloc(&devPtr, N * sizeof(float));
    cudaMemcpy(devPtr, hostPtr, N * sizeof(float), cudaMemcpyHostToDevice);
    ...
    simple_kernel<<<BLOCKS, BLOCK_SIZE>>>(devPtr, N);
```

```
    ...
    cudaMemcpy(hostPtr, devPtr, N * sizeof(float), cudaMemcpyDeviceToHost);
    cudaDestroy(devPtr);
    ...
}
```

4.1.2 CUDA Kernel

A kernel is similar to a conventional function we are used to from sequential programming. The main difference being it is executed on the GPU. We cannot, however, call a kernel in the same way as we would a regular function in conventional programming. As we mentioned previously in this chapter the CPU and GPU do not share memory. Calling a kernel with data only accessible from the CPU as an argument causes our program to crash. Before we can do work on the GPU, data has to be explicitly transferred from the CPU to the GPU. Additionally, kernels take some extra arguments. These arguments determine how many blocks and threads should be executed in parallel. A thread is the execution path of the kernel. They are divided into blocks, where each block is executed independently of others. Furthermore, these blocks are divided into a grid, which contains all threads launched by a kernel. Each thread in a kernel has an identical execution path but handles different parts of the input data.

4.1.3 Thread Hierarchy

To understand how Nvidia's GPUs execute in parallel, we need to understand what happens when we invoke a kernel. The arguments we give inside the brackets of a kernel-call lets the GPU know how many *threads* should be invoked. These threads are organized in a grid-structure as shown in figure 4.1. A grid represents a kernel-launch, and its size depends on the arguments given for the launch. The grid is divided into blocks, where the threads are contained. Threads are executed in a *warp*, a collection of 32 threads. A lot of the resource optimizations are based around warps. It is always a good idea to divide work into warps, as failing to do so keeps threads *idle*. Threads that are idle do no work, while also keeping resources occupied for other threads. Therefore, it is common practice to divide blocks into multiplicands of 32.

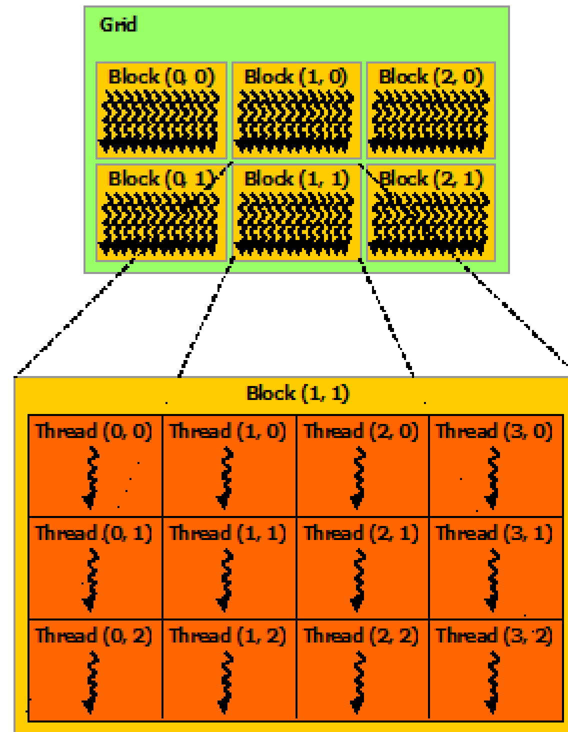


Figure 4.1: Thread Hierarchy, Reprinted from[18]

4.1.4 Memory Hierarchy

When we compute on the GPU, we have to manage resources more carefully than we would computing on the CPU. We do not think about memory stores and loads when writing CPU code, this is managed by the Operating System automatically. For GPUs, we do not have an operating system to do this for us. Programmers have to manage memory themselves, which adds another layer to coding. This can be challenging for beginners, but it can be a useful tool when we seek to optimize our programs' performance.

At the top level, we have the global memory. It has a lot of space, but low throughput. On block-level, we have shared memory which is comparable to the CPU's cache. It is smaller than the global memory but has much better *throughput*. Additionally, each thread has its own local memory that is much smaller but provides the best *throughput*.

The GPU's memory is organized in a hierarchy as shown in figure 4.2.

The CPU and GPU do not share memory. This means all input data has to be transferred from the CPU to the GPU, as the GPU does not support user input. Initially, all data is stored in global memory. Firstly, we have to allocate the memory needed on the GPU, similar to the `malloc()` function used in C. The function CUDA provides for this is simply called `cudaMalloc()`. From a programmers perspective, it is the exact same thing as `malloc()`

from C. Then, data has to be explicitly transferred from the CPU to the GPU.

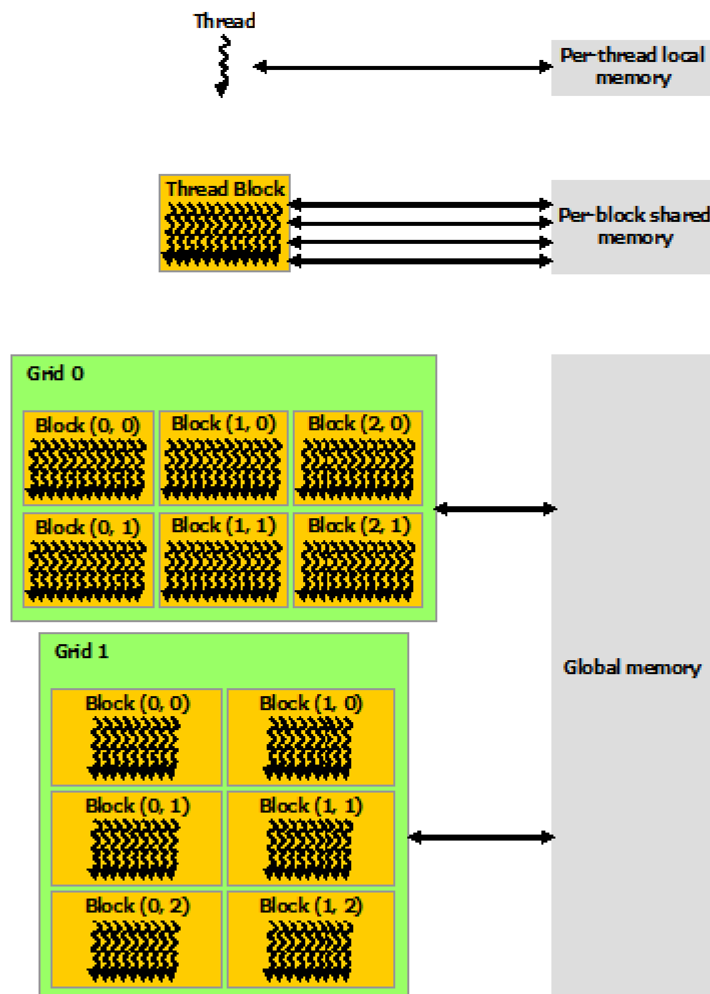


Figure 4.2: Memory Hierarchy, Reprinted from[18]

4.2 Libraries

In most programming languages many tools exist to help programmers implement fast and reliable code. Nvidia CUDA is no exception. To help speed up the process, and make our implementation more reliable, we have made use of some functions from external and internal libraries.

4.2.1 Thrust

Thrust was originally created by Jared Hoberock and Nathan Bell [8]. It has since evolved to be integrated into the CUDA toolkit, even though it was not specifically made for CUDA.

In addition to CUDA, Thrust provides overhead to numerous parallel frameworks such as OpenMP and TBB. The functions we will be using from this library is `inclusive_scan()` and `max_element()`. Intuitively, `max_element()` finds the max element from a sequence of numbers. An inclusive scan is executed over a sequence of numbers, known as an *array* of numbers in computer science. For each position, p in an array A, add each number from previous positions of A to the number at position p. So, for position p_i we have $p_i + p_{i-1} + p_{i-2} + ...p_0$ assuming no other positions have been computed.

In figure 4.3

1	2	3	4	5

1	3	6	10	15

Figure 4.3: Simple example of inclusive sum

4.2.2 CUB

CUB stands for CUDA UnBound and was created by Duane Merrill [16]. Contrary to Thrust, CUB is only optimized for CUDA C++. The functions we will be using from CUB is found in the classes `BlockScan()` and `BlockReduce()`. From the class `BlockScan()` we will be using `inclusive_sum()`, and from the class `BlockReduce()` we will be using reduce() and max(). These classes function on the block level, meaning they dedicate an entire thread block for execution.

Chapter 5

K-Core Decomposition

In this chapter, we present an algorithm for creating a k-core decomposition of undirected graphs. We parallelize this algorithm utilizing the Nvidia CUDA platform for heterogenous programming.

5.1 Our algorithm

Our algorithm is based on the graph property of locality, the neighborhood of vertices. By looking at nodes' adjacent k-cores, we can determine what k-core we have support for. We start by initializing every node's k-core to be equal to its degree. Node u is part of k-core G_k when u has at least k adjacent nodes with k-core larger or equal to k. We can count how many of the adjacent nodes have k-core larger than or equal to k, and if we count k or more nodes we know node u is in G_k. If we count less than k, u is not part of G_k. When this happens we need to update the k-core of u. The easiest way is to set k equal to k-1 and repeat the count with k-1 as our threshold. However, this potentially leads to extra steps, as we may need to reduce the k-core significantly more than 1. Therefore, we use a more efficient way of determining our next iteration of node u's k-core.

We know that the next iteration of the k-core of node u cannot be larger than our previous k. This means it has to be somewhere between our current iteration of k and the arbitrary case of 1. Instead of counting just the adjacent k-cores larger than or equal to k, we count how many we have of each adjacent k-core. By doing this we can determine which k-core node u has support for.

To find the next iteration of the k-core we utilize buckets in the form of a histogram. Each bucket, or column in the histogram, represents a number between k and 1. If an adjacent nodes' k-core k_{adj} is larger than or equal to k, add one to the first column. If $k_{adj} = k-1$

add one to the second column, if $k_{adj} = k - 2$ add one to the third, and so on. We now have buckets where their size corresponds to the number of adjacent k-cores, but this does not tell us which k-cores node u is part of. As with the case of having k or more nodes with k-core larger than or equal to k, we know buckets with size larger than or equal to the k-core it represents is supported. However, each bucket represents only one specific k-core, and not the k-cores larger. These k-cores also provide support. Because subgraphs of k-cores are nested, any node in k-core G_k is in $G_{k'}$ where k' is any k-core smaller than k.

As an example, bucket 5 represent k-core $k-5$, but has size c where $c < k-5$. By our logic k-core $k-5$ is not supported. However, lets say bucket 4, representing k-core $k-4$, has size $k-5$. This is not enough to provide support for k-core $k-4$, as $k-4 > k-5$, but we have $k-5$ nodes with k-core equal to $k-4$. Since $k-5+c >= k-5$, we have enough support for k-core $k-5$.

To account for this we add the size of every bucket representing a larger k-core into buckets representing smaller k-cores. This procedure of adding values is known as *inclusive sum*. To find the largest supported k-core of node u, we need to find the bucket representing the largest k-core providing support. The easiest way to do this is with a standard *linear search*, where we compare the size of a bucket to the k-core it represents starting from the largest to smallest. The first bucket we find where the size of it is larger than the k-core it represents will be our next iteration of k. *Linear search* is by no means optimal, as the similar *binary search* operates much faster. However, in a parallel setting searching it is best to use a *reduction-*function, which we will explain in an upcoming section.

Algorithm 3 Create a histogram

```
1: function MAKE_BUCKETS(node v)
2:     bins ← array with size k(v)
3:     for all u ∈ N(v) do
4:         adj_core ← k(u)
5:         if adj_core > k(v) then
6:             adj_core ← k(v)
7:         end if
8:         bins[k(v) - adj_core] ← bins[k(v) - adj_core] + 1
9:     end for
10: end function
```

In figure 5.1 node v has k-core $k = 10$. The number inside its adjacent nodes represent their respective current k-core estimate.

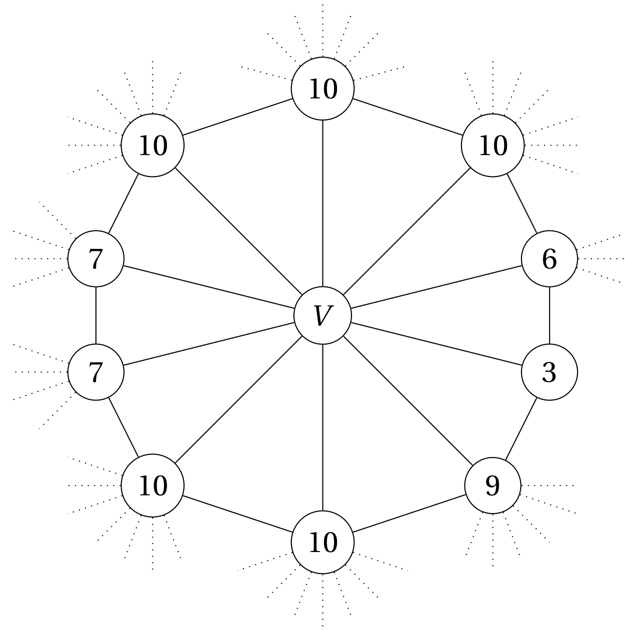


Figure 5.1: Neighborhood of vertex v

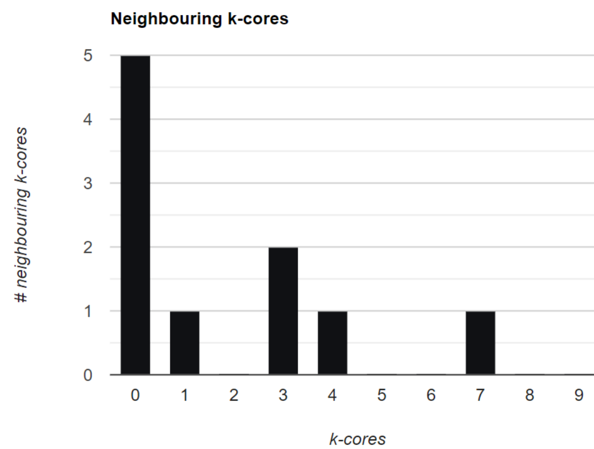


Figure 5.2: Result of make_buckets

Now that we have made a histogram of the adjacent k-cores of v, we can calculate the largest possible k we can support. We know any k-core larger than or equal to k, provides support for it. Furthermore, our histogram is sorted with the largest k-cores first, meaning any k from a previous position provides support. To account for this we can do an inclusive sum over the array. In an inclusive sum algorithm over array A, we add the sum from each previous position into the current position.

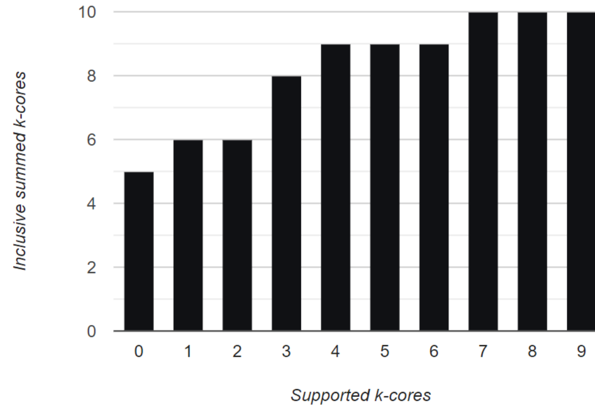


Figure 5.3: Result of the inclusive sum

When the inclusive sum algorithm is finished, we have an array with values in increasing order. Now we need to determine the largest k-core that has support using this array. We can determine this with an observation. Observe that the indexes of the array correspond to the k-core values between our old k-core and zero. Furthermore, each value at the index corresponds to the number of adjacent nodes that provide support. Keep in mind we flipped the position of our k-cores, which means at index $i = 0$ we have the previous k-core k, at index $i = 1$ we have $k - 1$ and so on. This means the value $v \in A$ at position i provides support for the k-core $k - i$. In our example above the value v at index $i = 5$ is 9, which means we have 9 adjacent nodes that has k-core either larger than or equal to $k - 5 = 5$. Therefore, k-core 5 is supported. We see that any $k - i <= v$ is a supported k-core. The largest k_{new}, where $k_{new} = k - i <= v$, is the largest supported k-core and the next iteration of node v's k-core.

Algorithm 4 Find the next k-core of a node

```
1: function NEXT_K-CORE(bucket B)
2:     k-core ← |size of B| - 1
3:     for all value v ∈ B do
4:         if v >= k-core then
5:             return v
6:         end if
7:         k-core ← k-core - 1
8:     end for
9: end function
```

Figure 5.4 shows the k-core estimates v is part of. We are interested in the largest one, which is 7 in this case.

Putting the procedures together, we get Algorithm 5.

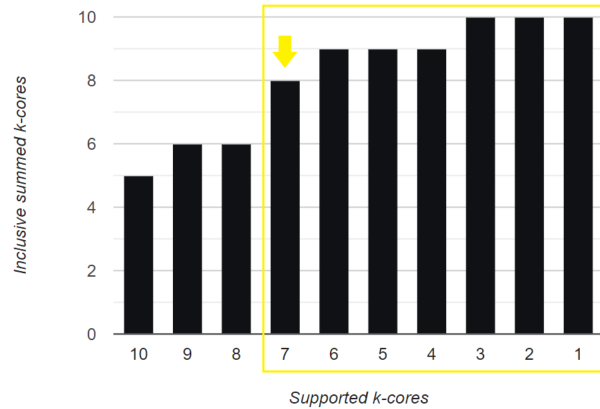


Figure 5.4: Possible K-cores of v

Algorithm 5 Next iteration of k-cores in Graph G

1: **function** DECOMPOSITION(Graph $G = (V, E)$)
2:     **for all** node v ∈ V **do**
3:         buckets ← make_buckets(v) (Algorithm 3)
4:         buckets ← inclusive_sum
5:         k(v) ← next_k-core(buckets (Algorithm 4)
6:     **end for**
7: **end function**

Unfortunately, running algorithm 5 once will not provide the correct k-core decomposition. When updating a node's k-core, the support it provided for its adjacent nodes is no longer there. In the figure 5.1 this means every adjacent vertex u with of vertex v with $deg(u) > 7$, no longer has the support of v. Furthermore, we cannot guarantee any of the supportive nodes in $N(v)$ updates to a lesser k-core in the same iteration. When this happens for enough nodes in $N(v)$, $v's$ current k-core will no longer be supported and need to update its k-core again. We ensure correctness by running the algorithm until none of the nodes in the graph updates its k-core as shown in Algorithm 6. When the algorithm completes, i.e. all nodes in G have converged, the *coreness* is found for all nodes.

Algorithm 6 k-core decomposition of Graph G

```
1:  function K-CORE_DECOMPOSITION(Graph G = (V, E))
2:      update ← true
3:      while update do
4:          update ← false
5:          for all node v ∈ V do
6:              prev_k ← k(v)
7:              buckets ← make_buckets(v) (Algorithm 3)
8:              buckets ← inclusive_sum
9:              k(v) ← next_k-core(buckets (Algorithm 4)
10:             if prev_k > k(v) then
11:                 update ← true
12:             end if
13:         end for
14:     end while
15: end function
```

5.1.1 Extended with CUDA

When computing in parallel it is necessary to think about our problem differently. Not only do we need to solve our problem, we also need to utilize as much of the available resources as possible. Neglecting to do this means losing out of potential huge time saves.

One approach to solving the k-core decomposition with CUDA is to dedicate an entire thread block to each vertex. As shown in algorithm 7, we run a kernel with N blocks (recall that N represents the number of vertices in a graph) where each block has 128 threads until update become false. When the value update is false none all vertices in graph G have converged to their coreness.

Algorithm 7 K-core decomposition host side

1: **function** K_CORE_DECOMPOSITION(Graph $G = (V, E)$)
2: $G_d \leftarrow$ graph G on device
3: N $\leftarrow$ number of vertices in G
4: update $\leftarrow$ true
5: update$_d \leftarrow$ update on device
6: **while** update **do**
7: next_k_core_estimates< < <N, 128> > >(G_d, updated)
8: update $\leftarrow$ updated
9: **end while**
10: G $\leftarrow G_d$
11: **end function**

The device side of a kernel demonstrating this is shown in algorithm 8. `Buckets` (line 7 of algorithm 8) is a *shared* array. A shared variable is stored at the block level, described in chapter 4.1.4. This means every thread contained in the block can access it. From line 9 to line 18 functions in the same way as described in algorithm 3. However, an added variable called `limit` is sometimes needed. When the k-core estimate of vertex v is larger than 128, the purpose `limit` is to place k-cores of adjacent vertices in the interval $[0, 127]$, which corresponds the indexes of `buckets`. This is required as the block primitives, described in chapter 4.2.2, utilizes exactly the number of threads in a block for execution. We calculate the limit for a vertex v with the formula:

$$limit = \left\lfloor \frac{k(v)}{128} \right\rfloor * 128 - 1.$$

The rest of algorithm 8 mimics algorithm 5.

Algorithm 8 Calculating next k-core estimate on device

```
1:  function NEXT_K_CORE(Graph G = (V, E), update)
2:      block_id ← unique ID of a block
3:      block_sz ← 128
4:      node v ← V(block_id)
5:      prev_k ← k(v)
6:      thread_id ← unique ID of a thread
7:      buckets[] ← shared array of size block_sz
8:      limit ← threshold for interval
9:      while thread_id < |N(v)| do
10:         if k(thread_id) >= limit + (block_sz - 1) then
11:             buckets[0] ← buckets[0] + 1
12:         else if k(thread_id) <= limit then
13:             buckets[(block_sz - 1)] ← buckets[(block_sz - 1)] + 1
14:         else
15:             buckets[k(thread_id) - limit] ← buckets[k(thread_id) - limit] + 1
16:         end if
17:         thread_id ← thread_id + block_sz
18:     end while
19:     buckets ← inclusive_sum(buckets)
20:     supported_k ← 128 - thread_id + limit
21:     if buckets[thread_id] >= supported_k then
22:         next_k ← supported_k
23:     else
24:         next_k ← 0
25:     end if
26:     max_supported_k ← BlockReduce(next_k, max())
27:     if max_supported_k < prev_k then
28:         update ← true
29:         k(v) ← max_supported_k
30:     end if
31: end function
```

In algorithm 8, we dedicate 128 threads to each vertex in a graph. However, in many real-world graphs vertices have a wide variety of degrees. In the graphs examined in chapter 6 some vertices have many neighbors, while some vertices have few neighbors. This means a vertex v with $deg(v) < 128$ will have *idle threads*, threads that do nothing while occupy-

ing resources. When computing in parallel we wish to achieve *load balancing*, every thread should do roughly the same amount of work. One possible solution is to execute our kernel with less than 128 threads, for example, 32 threads. This means vertices with degree less than 32 have idle threads. To keep all threads from idling, we would have to launch our kernel with 1 thread per block. While this is feasible, it is not the most efficient approach. Each streaming multiprocessor on the device executes one *warp*, 32 threads, in parallel, meaning we lose out on performance anyway.

Instead, we work towards load balancing by partitioning the vertices in a graph based on their degree. Each partitioned group of vertices is executed on a separate kernel, tuned to the degrees of the vertices. The interval for which vertices are partitioned is shown in table 5.1. Vertices placed in singles have $deg(v) = 1$, vertices placed in tiny have $2 <= deg(v) <= 7$, vertices placed in half_warp have $8 <= deg(v) <= 31$ and so on.

Partition	degree interval
singles	1
tiny	2, 7
half_warp	8, 15
one_warp	16, 31
two_warp	32, 63
four_warp	64, 127
eight_warp	128, 255
large	256 , maxDegree

Table 5.1: Partitioning of vertices based on degree

The single partition is ignored in the algorithm, as these vertices have achieved their coreness of 1. For partitions of vertices with small degree, we dedicate one thread block to multiple nodes. For partitions of vertices with large degree, we dedicate one thread block to each vertex. Each kernel performs the same as the kernel described in algorithm 8, but are tuned towards the degrees found in the partitions. Algorithm 9 shows how kernels are invoked on the CPU.

Algorithm 9 K-core decomposition host side

1: **function** K_CORE_DECOMPOSITION(Graph $G = (V, E)$)
2: G ← partitioned into intervals described in table 5.1
3: G_d ← graph G on device
4: update ← true
5: update[] ← array of updates on device for each partition group
6: **while** update **do**
7: tiny_k_core_estimates< < <TINY_BLOCK_SIZE, 1024> > >(G_d, update[0])
8: half_warp_k_core_estimates< < <HALF_WARP_BLOCK_SIZE, 1024> > >(G_d, update[1])
9: one_warp_k_core_estimates< < <ONE_WARP_BLOCK_SIZE, 1024> > >(G_d, update[2])
10: two_warp_k_core_estimates< < <|two_warp|, 32> > >(G_d, update[3])
11: four_warp_k_core_estimates< < <|four_warp|, 64> > >(G_d, update[4])
12: eight_warp_k_core_estimates< < <|eight_warp|, 128> > >(G_d, update[5])
13: large_k_core_estimates< < <|large|, 256> > >(G_d, update[6])
14: **if** any val ∈ update[] = true **then**
15: update ← true
16: **else**
17: update ← false
18: **end if**
19: **end while**
20: G ← G_d
21: **end function**

Chapter 6

Results

In this chapter, we present the processing time achieved from our k-core decomposition algorithm. We examine the run time of our kernels and discuss why some perform better than others. Furthermore, we compare how the overall run time of our parallel algorithm compares to sequential algorithms for solving k-core decomposition.

6.1 Hardware Used

NVIDIA Tesla V100 [21] was the GPU used to achieve the execution times presented. Tesla V100 offers the performance of up to 100 CPUs, enabling computing challenges once thought impossible. This GPU is built with the Volta architecture [20]. It is one of world's most advanced data center GPUs ever built to accelerate Artificial Intelligence(AI), High-Performance computing(HPC), and graphics.

6.2 Graphs

All the graphs we use to analyze are undirected and unweighted. Making a correct algorithm for other types of graphs require a bit more care, but the overall idea stays the same. For this reason we have chosen to keep graphs simple, focusing instead on optimizing the runtime of kernels. The graphs used to examine the runtime of our implementation can be found at Stanford network analysis project [11].

	Nodes	Edges	highest degree
Facebook-combined	4039	88 234	1045
As-skitter	1 696 415	11 095 298	35 455
Wiki-topcats	1 791 489	28 511 807	238 607

Table 6.1: Overview of graphs used to examine runtimes

6.3 Timing method

We have chosen to ignore the time taken to transfer data from the CPU to the GPU, and instead focus on the kernel runtimes. The time taken to transfer data is by no means uninteresting, but most of the speedup of implementations can be achieved by optimizing kernels. We also present each kernel's runtime and some information of intermediate steps. The configuration of kernels is explained in chapter 5.1.1, where each kernel execute on one partition group.

6.4 Execution times

6.4.1 Total runtimes

Table 6.2 shows the combined runtime of all kernels for each graph. Iterations represent the number of estimates were needed until all vertices converged to their coreness. Note that runtimes are presented in milliseconds.

Graph	Iterations	Runtime in ms
facebook-combined	18	1.48
as-skitter	39	38.8
wiki-topcats	62	156.8

Table 6.2: Parallel algorithm with CUDA

Table 6.3 shows the runtime of the comparable sequential algorithm described in chapter 5.1. Iterations represent the number of estimates that were needed until all vertices converged to their coreness. Note that runtimes are presented in milliseconds.

Graph	Iterations	Runtime in ms
facebook-combined	12	12.20
as-skitter	34	5652.10
wiki-topcats	52	18 346.90

Table 6.3: Comparable sequential algorithm

Table 6.4 shows the runtime of a faster sequential algorithm based on ideas described in chapter 2. Note that runtimes are presented in milliseconds.

Graph	Runtime in ms
facebook-combined	2.44
as-skitter	958.13
wiki-topcats	2 186.65

Table 6.4: Fast sequential algorithm

Table 6.5 shows the speedup achieved with the parallel algorithm compared to the sequential algorithms. *Speedup Comparable* is the speedup versus a comparable sequential algorithm, *Speedup Fast* is the speedup compared to a faster sequential algorithm. The speedup is calculated with the formula $speedup = runtime_{seq}/runtime_{par}$. So, if $runtime_{seq} = 1000ms$ and $runtime_{par} = 100ms$ we have $speedup = 1000ms/100ms = 10$, which means the runtime of the parallel execution is ten times faster compared to the sequential execution.

Graph	Speedup Comparable	Speedup Fast
facebook-combined	8.24	1.65
as-skitter	145.67	24.69
wiki-topcats	117.01	13.95

Table 6.5: Speedup Parallel vs Sequential

6.4.2 Kernel runtimes

For each of the following tables *Kernel* represent kernels tuned to the partitions as explained in chapter 5.1.1. *Nodes* is the number of vertices in a partition group, *Iterations* are the number of estimates of k-cores in a partition group until all vertices have converged to coreness. Note that runtimes are presented in milliseconds.

Kernel	Nodes	Iterations	Runtime in ms
tiny	579	3	0.16
half	741	7	0.17
one	907	18	0.18
two	835	18	0.16
four	597	17	0.16
eight	298	16	0.16
large	7	16	0.16

Table 6.6: Runtime of each kernel for facebook-combined

Kernel	Nodes	Iterations	Runtime in ms
tiny	903 193	9	8.30
half	318 795	17	7.20
one	149 829	17	8.90
two	68 987	18	4.70
four	21 819	37	2.10
eight	9364	39	1.63
large	6029	38	5.35

Table 6.7: Runtime of each kernel for as-skitter

Kernel	Nodes	Iterations	Runtime in ms
tiny	489 255	24	7.95
half	527 400	19	18.65
one	412 126	20	37.25
two	214 803	28	21.35
four	90 646	61	12.25
eight	35 922	62	9.23
large	21 337	62	52.60

Table 6.8: Runtime of each kernel for wiki-topcats

6.5 Discussion

From the runtimes of kernels obtained in section 6.4.2 we notice a considerable discrepancy of runtimes in the graphs *as-skitter* and *wiki-topcats*. There seems to be no correlation between the number of k-core estimates and runtimes. This can easily be explained. The k-core estimates needed for all vertices in a partition group to achieve coreness do not represent the number of computations performed. We cannot guarantee that k-core estimates of a partition group are influenced by vertices in other groups. Indeed, we found some partition group that had seemingly converged needing updates in the next iteration. The runtimes of kernels for *facebook-combined* are too similar to draw any conclusion from.

The speedup obtained in section 6.4.1 is decent for the parallel execution. Not surprisingly, the parallel algorithm is much faster than the comparable algorithm in sequential. It is also faster than the fast approach in sequential, albeit not as considerably. However, the speedup is smaller for the graph *wiki-topcats* compared to the graph *as-skitter* even though it is larger. This could be explained by the number of iterations needed. Furthermore, the large kernel has noticeably worse runtime compared to other kernels for the wiki-topcats. For the graph *as-skitter* this is not the case. This can be explained by the number of vertices in the partitioning of large, as well as the considerably higher max degree of *wiki-topcats* (238 607) compared to *as-skitter* (35 455). It seems the highest degree of a graph has a larger impact on the implementation in parallel compared to sequential. This can be explained by the number of estimates needed before large k-cores converge. During each step, we iterate through each adjacent node of a node v, meaning a substantially higher amount of iterations is needed for *wiki-topcats* compared to *as-skitter*.

Chapter 7

Conclusion

The purpose of this thesis was to explore how an algorithm based on the *locality*-property, neighborhood of vertices, could perform in parallel with Nvidia CUDA. The algorithm we implemented did not necessarily intend to be the fastest available, but the nature of its execution suits a parallel approach well. The recursively exhaustive algorithm [3] performs excellently in general, but it does not translate well to parallel execution. For some values of k-core k, we may need to remove only a few vertices. When we compute for these values, a lot of the resources of parallel systems are left unused.

The idea behind our approach was to keep threads from idling, thus utilizing resources more efficiently. When we compute based on the locality, we can calculate every node simultaneously. We can do this because each node computes independently of the others. Each vertex depends on the k-core of adjacent ones, but for each intermediate step, it is sufficient to use the previous k-core. Furthermore, *race conditions* work in the algorithms favor. Some iterations of node u may be skipped if some of its adjacent nodes calculate their new k-core faster, thus influencing the result of the computation of $k(u)$. This does not produce an incorrect result, as this computation would have taken place in the next iteration either way.

From the results obtained in this thesis, we found the parallel algorithm to be much faster than the sequential one. However, managing resources proved to be difficult even with our approach. For some intermediate iterations, especially as it approaches finished, only a few nodes' k-core is computed. This is natural as nodes with smaller degree reach their maximum k-core in fewer iterations than nodes with a larger degree.

7.1 Further Work

Treating the algorithm presented in this thesis as an approximation algorithm, could yield interesting results. Rather than focusing on a correct result, we could sacrifice accuracy for performance. We could extend the implementation to larger data sets, too large to fit on one disk. The nature of the algorithm supports localized execution on graphs.

Another approach is to combine algorithm techniques to potentially improve execution times. Instead of an algorithm tailored purely towards the *locality property*, we could use the exhaustive approach discussed in chapter 2 to preprocess graphs.

Bibliography

[1] José Ignacio Alvarez-Hamelin et al. "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases". In: *arXiv preprint cs/0511007* (2005).

[2] Gary D Bader and Christopher WV Hogue. "Analyzing yeast protein–protein interaction data obtained from different sources". In: *Nature biotechnology* 20.10 (2002), pp. 991–997.

[3] Vladimir Batagelj and Matjaž Zaveršnik. "Fast algorithms for determining (generalized) core groups in social networks". In: *Advances in Data Analysis and Classification* 5.2 (2011), pp. 129–145.

[4] Sergey N Dorogovtsev, Alexander V Goltsev, and Jose Ferreira F Mendes. "K-core organization of complex networks". In: *Physical review letters* 96.4 (2006), p. 040601.

[5] Michael J Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[6] Linton C Freeman. "Centrality in social networks conceptual clarification". In: *Social networks* 1.3 (1978), pp. 215–239.

[7] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[8] Jared Hoberock and Nathan Bell. *Thrust*. 2009. URL: https://thrust.github.io/.

[9] Intel. *Intel Xeon W-3275 Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/193752/intel-xeon-w-3275-processor-38-5m-cache-2-50-ghz.html.

[10] Maksim Kitsak et al. "Identification of influential spreaders in complex networks". In: *Nature physics* 6.11 (2010), pp. 888–893.

[11] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[12] Don R. Lick and Arthur T. White. "k-Degenerate Graphs". In: *Canadian Journal of Mathematics* 22.5 (1970), pp. 1082–1096. DOI: 10.4153/CJM-1970-125-1.

[13] David Luebke and Greg Humphreys. “How gpus work”. In: *Computer* 40.2 (2007), pp. 96–100.

[14] David W Matula and Leland L Beck. “Smallest-last ordering and clustering and graph coloring algorithms”. In: *Journal of the ACM (JACM)* 30.3 (1983), pp. 417–427.

[15] Chris McClanahan. “History and evolution of gpu architecture”. In: *A Survey Paper* 9 (2010).

[16] Duane Merrill. *CUB*. 2011. URL: https://nvlabs.github.io/cub/.

[17] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. “Distributed k-core decomposition”. In: *IEEE Transactions on parallel and distributed systems* 24.2 (2012), pp. 288–300.

[18] NVIDIA. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[19] Nvidia. *Deep Learning*. URL: https://developer.nvidia.com/deep-learning.

[20] Nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[21] Nvidia. *Tesla V100*. 2018. URL: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[22] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011. ISBN: 9780123742605 0123742609.

[23] Stephen B Seidman. “Network structure and minimum degree”. In: *Social networks* 5.3 (1983), pp. 269–287.