A naive implementation of Topological Sort on GPU

A comparative study between CPU and GPU performance

MARTIN EKLUND

DAVID SVANTESSON
A naive implementation of Topological sort on GPU

A comparative study between CPU and GPU performance

MARTIN EKLUND
DAVID SVANTESSON

Degree Project in Computer Science, DD143X
Supervisor: Mårten Björkman
Examiner: Örjan Ekeberg

CSC, KTH, 2016
Abstract

Topological sorting is a graph problem encountered in various different areas in computer science. Many graph problems have benefited from execution on a GPU rather than a CPU due to the GPU's capability for parallelism. The purpose of this report is to determine if topological sorting may benefit from a naive implementation on the GPU compared to the CPU. This is accomplished by constructing a parallel implementation using the CUDA platform by NVIDIA for GPGPU programing. The runtime of this implementation running on several different graphs is compared to a sequential implementation in C running on the CPU. The results indicate that the GPU algorithm only works beneficially on large, shallow graphs.
Sammanfattning

En naiv implementation av topologisk sortering på GPU
En jämförande studie mellan CPU och GPU prestanda

Topologisk sortering är ett grafproblem som förekommer i ett flertal olika områden inom datavetenskap. Många grafbaserade problem gynnas av att exekveras på GPU istället för CPU på grund av GPU:ns förmåga till parallellism. Syftet med denna rapport är att avgöra om topologisk sortering kan gynnas av en naiv implementation på GPU istället för att implementeras på CPU. Detta utförs genom att konstruera en parallell implementation i NVIDIA:s GPGPU-plattform CUDA. Körtiden för denna implementation som körs på ett flertal olika grafer jämförs med en sekventiell implementation skriven i C som körs på CPU. Resultaten visar att en GPU implementation endast är fördelaktig när stora, grunda grafer behandlas.
# Table of Contents

1. Introduction .................................................................................................................. 1  
   1.1 Problem Statement ................................................................................................. 1  
   1.2 Scope .................................................................................................................. 2  
   1.3 Motivation ........................................................................................................... 2  

2. Background .................................................................................................................. 3  
   2.1 Topological Sort Explained ................................................................................... 3  
   2.2 Prior Studies on Topological Sort on GPU ............................................................ 4  
   2.3 Kahn’s Serial Algorithm ......................................................................................... 4  
   2.4 GPGPU Programming .......................................................................................... 5  
   2.5 GPU vs CPU Performance ..................................................................................... 5  
   2.6 GPU Architecture .................................................................................................. 6  
   2.7 CUDA Programming ............................................................................................. 7  

3. Method ........................................................................................................................ 10  
   3.1 Generating Data .................................................................................................... 10  
   3.2 Data Structures ..................................................................................................... 11  
   3.3 GPU Implementation of Kahn’s algorithm ............................................................. 12  
   3.4 Performance Measurements .................................................................................. 13  
   3.5 Verifying Data ...................................................................................................... 13  
   3.6 Analyzing Data .................................................................................................... 13  
   3.7 Hardware and Software Used ............................................................................... 14  

4. Results ......................................................................................................................... 15  

5. Discussion .................................................................................................................... 19  
   5.1 Analysis of Result .................................................................................................. 19  
   5.2 Code Analysis ........................................................................................................ 21  
   5.3 Future Work ........................................................................................................... 22  

6. Conclusion ................................................................................................................... 23  

7. References ................................................................................................................... 24  

GPU Pseudo Code Appendix ......................................................................................... 27
1. Introduction

Many graph algorithms are known to be sequential in nature, therefore being unsuitable for implementation on the GPU. However research has shown that solutions to these types of graph problems may benefit from parallel implementations. The last decade has seen a growing trend towards implementing algorithms on GPUs in order to increase performance, sparking an interest in the field of computer science [1].

Topological sorting is a classic graph problem, where the most used algorithms are of a sequential nature. It is the problem of constructing a linear ordering of the vertices in a DAG (directed acyclic graph). The rule of the linear ordering is as follows, given a DAG, every edge $uv$ from node $u$ to $v$, means that $u$ must come before $v$ in the linear ordering of the graph [2].

Topological sorting is an important aspect in the scheduling of jobs such as instruction scheduling, determining the order of compilation tasks to perform in makefiles, resolving symbol dependencies in linkers, and deciding in which order to load tables with foreign keys in databases. Most users of Debian-based UNIX systems have used topological sorting indirectly by invoking the apt-get command, which uses topological sorting in order to determine the best way to install or remove packages [3]. Another example of using topological sorting in research is using it to find the best ways to schedule a large amount of jobs in distributed networks [4].

This report aims to investigate whether it is possible to gain any performance by implementing a naive parallel implementation of topological sorting on a GPU rather than a CPU. The parallel implementation is constructed using CUDA, a parallel programming API developed by NVIDIA to use with their graphics cards.

1.1 Problem Statement

The key purpose of this study is to determine whether an implementation of Kahn’s serial algorithm for topological sorting algorithm may benefit in performance from parallelization, by being ported from the CPU to the GPU. In order to investigate this aim, this study measures the execution time of the parallelized implementation executed on the GPU. These results are then compared to the execution time of the corresponding serialized implementation executed on the CPU.
1.2 Scope

The parallel GPU implementation consists of an algorithm similar to the sequential version. It is naive in the sense that it keeps the logic of the algorithm as close to the original CPU version as possible. All benchmarks are conducted on computers using hardware specifications typical for normal home PCs. This thesis does not engage in investigating the existence of any optimal algorithms. The reader should bear in mind that the implementation is constructed in CUDA only, and limitations concerning the platform may exist when comparing it to other GPGPU (general-purpose computing on graphics processing units) platforms.

1.3 Motivation

By speeding up algorithms through GPGPU there might be a benefit when handling big and complex graphs, making them solvable in a reasonable amount of time. Using algorithms that take advantage of parallelization could give an advantage over previous serial implementations where calculating too large graphs would not be practically applicable.

Our main motivation is to find out if a naive parallel implementation of Kahn’s serial algorithm can be sped up by GPGPU using CUDA. Another motivation is personal interest in the expanding area of GPGPU computing.
2. Background

2.1 Topological Sort Explained

Given a directed and unweighted Graph \( G = (V, E) \), creating a topological sort of the graph is the process of ordering the vertices \( V \) in a way such that if the edge \( uv \) exists between node \( u \) and \( v \), \( u \) comes before \( v \) in the sorted set [5]. This can be defined as a way of scheduling a set of tasks with hard dependencies between each other such that all dependencies for a task are solved before executing said task.

![Directed Acyclic Graph](image)

**Figure 1.1:** Directed Acyclic Graph [6]

A topological sort divides the graph into layers of nodes that can be executed freely if connected nodes from all layers above have been executed prior. There is no guarantee of the order within each layer, and neither does it matter, because all that is being sought after is vertices which belong to the same group where all prior dependencies have been fulfilled.

There exist some conditions that must be met by the graph in order to allow a topological sort. The graph needs to be a DAG, meaning that the graph is directed and contains no cycles. If the graph contains a cycle, it will create a circular dependency causing a circular dependency set, making it impossible to order the vertices such that \( u \) always comes before \( v \) in the sorted set.
2.2 Prior Studies on Topological Sort on GPU

There is little published data on any implementations of solving a topological sorting problem using CUDA. However several papers have documented the use of parallel algorithms to solve the problem, but the majority of them are old and theoretical in nature [7][8][9]. The parallel algorithms in these research studies use an exponential amount of processors, making them impractical to implement on current GPU hardware.

The most common approach to performing a topological sort is to implement an algorithm based on either breadth first search (BFS) or depth first search (DFS). The DFS approach was first described by Tarjan in 1976 [10]. However research has shown that parallelization of DFS algorithms is extremely difficult and performance gains are limited and hard to achieve [11]. The BFS approach was first described by Kahn in 1962 [5]. Parallelizing a BFS implementation has proven to be a suitable solution for similar problems when using CUDA [11]. For this reason Kahn’s algorithm was chosen for this study.

However, studies conducted on graph algorithms in general have shown that graph problems are hard to implement on GPUs due to how the hardware is built. Memory access in graph algorithms are often highly irregular, resulting in badly coalesced accesses. This is often due to graph traversal in algorithms, where nodes connected by an edge might reside far from each other in the device memory [12]. Despite this, there are studies that have shown that running graph algorithms on a GPU can be beneficial [1].

2.3 Kahn’s Serial Algorithm

Kahn’s algorithm [5], summarized in pseudo code at the end of this section, works in linear time with $O = (|V|+|E|)$. Given a graph G, a starting set Q is created with all vertices in G that have no incoming edges. In each iteration, a node $n$ is popped from Q and put at the end of list $R$, and all edges connecting $n$ to any other vertices in G are removed. All vertices in G that now has no incoming edges are added to Q, and the loop continues. If the loop stops before G is empty the graph is not a DAG, and a topological sort is therefore impossible. The algorithm terminates when Q is empty at the start of an iteration.
Pseudo code for Kahn's algorithm

R ← Initialise empty list, the result list
Q ← Queue of all nodes with no incoming edges
G ← Graph containing all vertices and edges
Populate Q with all nodes in G that have no incoming edges, remove from G
while Q not empty:
   remove node n from Q
   add n to end of R
   for each node w with an edge e in E from n to w:
      remove edge e from G
      if w has no other incoming edges:
         insert w into Q
         remove w from G
if graph has edges:
   return error (graph has at least one cycle, graph is not a DAG)
else
   return R (a topologically sorted order)

2.4 GPGPU Programming

As mentioned, GPGPU is an abbreviation for general-purpose computing on graphics processing units. Graphic rendering usually requires hardware (a GPU) capable of performing single instructions on multiple data (SIMD) in order to be fast enough to render graphics in real time. As the name suggests, GPGPU means that the GPU (which normally only handles graphics computing) is used for general calculations that would typically be handled by the CPU. While CPU cores are generally much faster than GPU cores, the GPU has many more cores than the CPU, making GPGPU a suitable solution for problems where parallel solutions may be implemented.

2.5 GPU vs CPU Performance

According to NVIDIA, a team at NASA successfully reduced the analysis time of one of their programs for analysing traffic air flow from 10 minutes to 3 seconds with a GPGPU implementation using CUDA [13]. NVIDIA lists several other examples on their website, where speedups of up to 300 times have been achieved. A paper written by employees at Intel Corporation questions the accuracy of these results and claims that after optimizing programs for multi-core CPUs, the GPU is only 2.5 times as fast as the CPU on average [14]. NVIDIA’s response to this paper claimed that while 100 times speedups might not be possible for all programs, the real benefit of GPGPU programming is that it is not as complex to implement GPU implementations with CUDA as multi-core CPU implementations [13]. According to NVIDIA, a team at M.I.T had come to the conclusion that when comparing the time investment spent on both CPU and GPU implementations, the performance gain on the GPU was 35 times as much as on the CPU [13].
2.6 GPU Architecture

A modern GPU typically consist of several streaming multiprocessors (SM), each capable of running up to thousands of threads simultaneously. An SM contains several streaming processors (SP), each with their own registers. Each SP executes several threads at the same time using SIMD. A modern GPU is capable of running thousands of threads concurrently. Each SM has its own caches and shared memory, which is directly accessible from threads running on the same SM as shown in figure 2.1. The device memory (also known as global memory) shown in the same figure is available to all threads on the GPU. Access to this memory is roughly 100 times slower than accessing the shared memory [15].
2.7 CUDA Programming

CUDA is a parallel computing platform and programming model developed by NVIDIA that has become widely used for GPGPU programming [16]. Different NVIDIA GPU hardware has different CUDA compute capabilities, meaning that some CUDA functions are only available on the latest generation of NVIDIA GPUs. Developers can send high-level language code (such as C, C++ and Python) to the GPU when using CUDA. This eliminates the need to know how to implement parallel programs on an assembly language level, something researchers previously were forced to do in order to utilize the GPU hardware for research purposes [16].

![CUDA Programming Model](image)

**Figure 2.2** CUDA programming model [1]
Threads on the GPU are grouped into different hierarchies which can have between one and three dimensions that are typically set by developers. Threads are grouped into different blocks which are in turn grouped into a single grid (shown in figure 2.2). A block is run on one single SM until all the threads in the block have terminated. One SM can run several blocks concurrently. Within each block threads are divided into warps, and all threads within a warp is executed simultaneously in SIMD manner.

To execute code on the GPU using CUDA, the programmer needs to write a kernel that contains all the code to be run on the GPU. These kernels are then launched on the GPU (known as the device in CUDA terms) from a program running on the CPU (known as the host in CUDA terms). When launching a kernel it is possible to specify the dimension of the grid, the dimension of the blocks, and the number of threads each block should contain (represented in figure 2.2). These dimensions and sizes may affect the kernels performance on the GPU, depending on how algorithms and data is constructed. There is also a need to allocate memory on the device before launching kernels. This is accomplished with cudaMalloc which works similarly to the malloc function in C. When memory has been allocated on the device, data needs to be transferred from the host to the device. Transferring memory can be conducted both from the device to the host, and from the host to the device. These types of memory operations are expensive and should be avoided as much as possible [17].

A common problem in parallel programming is race conditions and the need for synchronization. CUDA offers synchronization between threads in a block, but not between threads being executed in different blocks. Another tool to avoid race conditions when reading and writing data is the atomic operations. These operations are guaranteed to write and read to addresses in either global or shared memory without interference from other threads [18]. However using atomic operations might have a negative impact on the performance of the kernel. If different threads attempt to perform an atomic operation on the same address, then those operations will be executed in a serialized fashion.

When an SM executes a block, it is not guaranteed to execute branches in the code in parallel. Different branches may be executed in parallel thanks to the instruction cache. However when conditional branches become too large, the SM might have to execute all the threads in which the condition holds true separately from those where it does not [12]. Therefore context switching takes place in the SM between the different SPs executing different branches. This is a problem that is difficult to avoid due to the need for branching in many programs.

Accessing global memory on the GPU works best when accessing memory locations lying close in the device memory. This is due to how the hardware accesses memory. The memory in hardware is constructed using segments of 32 to 128 bytes. When many threads in a warp access memory from the same segment it is called a coalesced memory access, and minimizes the amount of reads that has to be done
from the global memory. If all threads read or write from separate segments however, many more memory accesses have to be scheduled. This has great performance significance in algorithms, and to obtain maximum performance out of a GPU it is important to group read/write accesses in each kernel so that they lie close to each other in space and time [19].
3. Method

The parallel GPU implementation was constructed in CUDA C using CUDA version 7.5.19 since CUDA is widely used in the scientific community when it comes to GPGPU programming. The CPU implementation was constructed in C using the C99 standard, using only standard libraries. The language C was chosen because a CPU implementation that is similar to the GPU implementation with regards to properties, such as memory allocation, was desirable. Both implementations use the same data and data structures described in the following sections.

3.1 Generating Data

Data for this study was generated using the Stanford Network Analysis Platform (SNAP) library for python [20]. Through this library, a great number of directed graphs with specified amounts of both nodes and edges were generated in order to be able to determine if and how the number of nodes and edges affect the execution times of the CPU and GPU programs. The algorithm used in the library makes use of the Erdős-Rényi method for generating random graphs [21]. However, the algorithm used does not allow generation of DAGs, therefore the generated graphs had to be altered using the acyclic command available on Linux [22]. This command is able to take a directed, non-acyclic graph as input and produce a directed acyclic graph as output by reversing directions of certain edges in the graph that are part of a cycle. Unfortunately manually setting the depth (number of layers) of the graphs generated was not possible. In order to be able to test how different depths affected the implementations a simple C program was written. In this program a DAG with a specified number of layers and a specified amount of nodes which each layer would contain was constructed. Each node generated in this program had one incoming edge and one outgoing edge, apart from the layer at the very top and the layer at the very bottom.

No DAGs based on real world data were found. However, several large directed graphs based on real world networks were made acyclic using the same method as on the graphs generated with the SNAP library. These graphs were taken from the SNAP datasets [23].
3.2 Data Structures

Due to the data in most of the graphs being very sparse, the choice was made to represent the graph data using a compact adjacency list to be able to fit as large graphs as possible on both GPU and CPU [1]. Using this format, three arrays are created representing nodes, edges and the indegree for each node of the graph. The indegree array specifies how many incoming edges each node has. The node array \( V \) stores information about offset values for the edge array, where index of \( V \) corresponds to node number. The edge array \( E \) stores all edges in the whole graph in consecutive order, and the offset values in \( V \) are used to find all outgoing edges for nodes in \( V \). The relationship between these two arrays is depicted in figure 3.1 below. To obtain the edge count for a node in the node array the program only has to calculate the difference in \( V \) between node \( i \) and node \( i+1 \). A third array \( I \) stores the indegree for each node and is used in each iteration of the algorithm to determine which nodes to remove edges from, and to decide which nodes are to be added to the result list in each iteration.

![Diagram of node and edge arrays](image)

**Figure 3.1**: Correlation between node array \( V \) and edge array \( E \) [1]
3.3 GPU Implementation of Kahn’s algorithm

The GPU Implementation of Kahn’s algorithm was implemented using CUDA C. The program is structured with three device kernels and one outside loop on the host launching the kernels repeatedly until the algorithm terminates, as illustrated by figure 3.2. It works in a similar manner to the serial algorithm, using a global queue system. The enqueuer kernel takes care of searching the indegree array and populating the queue, it launches as many threads as there are nodes in the graph. The dequeuer kernel consumes the queue, removes edges by subtracting from the indegree array, and writes results to the result array. The dequeuer kernel launches as many threads as there are nodes in the queue. The reset queue kernel sets the queue value to zero before iterating again. The algorithm terminates when the queue is empty after having run the enqueuer kernel and is checked every iteration in the outside loop. The value for queue size is copied from device to host in every iteration of the loop. For more details, please refer to the full pseudo code included in the appendix.

Figure 3.2: Flow-chart of outer loop of GPU implementation
3.4 Performance Measurements

On the CPU, measurements regarding performance of the algorithm itself were taken, without taking the setup time for the graph in memory into consideration. These measurements were conducted using the CPU clock of the computer.

The GPU measurements were conducted in a similar fashion, only measuring the execution time of the algorithm itself without taking initialization time into account. These measurements were all taken using the GPU clock with the use of Cuda Events [24]. This is due to the fact that timings from the CPU clock might not be accurate when timing the GPU implementation. Instead it is recommended by NVIDIA to use Cuda Events to obtain accurate measurements.

3.5 Verifying Data

In order to ensure that both programs produced a topologically sorted list of a graph, a program was written in C that compares both resulting lists to a list generated by the tsort command available in Unix. The program takes three lists, one being the topologically sorted list produced by tsort and one being the topologically sorted list generated by the GPU implementation. The verification program also needs a list containing the number of nodes that each layer of the graph has, which is easy to produce in the GPU implementation. The verification program uses these three lists to ensure that both the number of nodes in each layer and the node indexes themselves correspond between the both topologically sorted lists. Once the GPU implementation was verified the program was executed again, only this time with the topologically sorted list generated by the CPU implementation in place of the list generated by the tsort command.

3.6 Analysing Data

When comparing the performance of the GPU and the CPU implementations, several major factors were taken into account. First, graphs of different properties were used under the assumption that the properties will greatly impact the performance. Three different plots were created to help illustrate what impacts performance in the algorithms. One plot describes the performance measured in time when changing node count, and keeping edges static. Another graph is plotted on edge count, keeping the node number static. Lastly, depth is measured against time, keeping both the node count and edge count static.
To make sure that accurate measurements of each graph type was obtained, five test files was generated for each graph type. This is done in order to avoid outliers in the randomly generated graphs. Further, each algorithm was executed five times for each test file and a simple statistical average was been taken. This is to ensure that proper time performance is realized since optimizations in CPU and GPU using caching might factor into performance.

### 3.7 Hardware and Software Used

The following tables show the hardware and software used for this study, these were used during benchmarking for all tests.

<table>
<thead>
<tr>
<th>Hardware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.3 GHz Intel Core i7</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce GT 750M 2GB, Compute Capability 3.0</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB 1600 MHz DDR3</td>
</tr>
</tbody>
</table>

**Table 3.1: Hardware**

<table>
<thead>
<tr>
<th>Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>OS X El Capitan 10.11.4</td>
</tr>
<tr>
<td>Graph Generator</td>
<td>Python 2.7.6 with SNAP 1.2</td>
</tr>
<tr>
<td>CPU implementation language</td>
<td>C with C99 Standard compiled with gcc 4.2.1</td>
</tr>
<tr>
<td>GPU Implementation Language</td>
<td>CUDA C, CUDA version 7.5.19</td>
</tr>
</tbody>
</table>

**Table 3.2: Software**
4. Results

The tables below illustrate benchmarks procured for both implementations, using the data generated. The depth in each table represents the depth of the graph being tested, and signifies the amount of layers created for each topological sort. Due to hardware limitations the maximum node and edge amount were limited to 2,000,000.

In table 4.1 and 4.2 each graph with the same characteristics has been generated five times by using the Erdős-Rényi function in the SNAP library, and a statistical average taken for depth and time in five consecutive benchmarks for each graph file. The average depth and time in each table represents the statistical average of depth taken between the five generated files of each type.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>Avg. Depth</th>
<th>Avg. Runtime CPU (ms)</th>
<th>Avg. Runtime GPU (ms)</th>
<th>Runtime ratio (CPU/GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^6$</td>
<td>$10^3$</td>
<td>1.4</td>
<td>140</td>
<td>8.4</td>
<td>16</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^4$</td>
<td>2.2</td>
<td>140</td>
<td>8.2</td>
<td>17</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^5$</td>
<td>5.6</td>
<td>150</td>
<td>9.2</td>
<td>16</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>340</td>
<td>290</td>
<td>120</td>
<td>2.5</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$2 \cdot 10^6$</td>
<td>25,000</td>
<td>310</td>
<td>7400</td>
<td>0.04</td>
</tr>
</tbody>
</table>

**Table 4.1:** Comparing runtime between CPU and GPU implementations on graphs with 1,000,000 nodes and varying amounts of edges.

![Figure 4.1](image.png)

**Figure 4.1:** Data from table 4.1 plotted on edges and runtime using logarithmic scales on both axes.
As table 4.1 show, the CPU runtime increases slightly when the amount of edges vary between 1,000-100,000 edges. This also holds true for the GPU implementation, where little difference is shown. When the number of edges increases to 2,000,000, the CPU runtime increases fairly while the GPU runtime increases considerably.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>Avg. Depth</th>
<th>Avg. Runtime CPU (ms)</th>
<th>Avg. Runtime GPU (ms)</th>
<th>Runtime ratio (CPU/GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>$10^5$</td>
<td>990</td>
<td>0.4</td>
<td>120</td>
<td>0.0033</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^5$</td>
<td>8,300</td>
<td>1</td>
<td>390</td>
<td>0.0026</td>
</tr>
<tr>
<td>$10^7$</td>
<td>$10^5$</td>
<td>190</td>
<td>16</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$10^8$</td>
<td>$10^5$</td>
<td>5.6</td>
<td>150</td>
<td>9.2</td>
<td>16</td>
</tr>
<tr>
<td>$2\times10^8$</td>
<td>$10^5$</td>
<td>4.0</td>
<td>280</td>
<td>21</td>
<td>14</td>
</tr>
</tbody>
</table>

**Table 4.2**: Comparing runtime between CPU and GPU implementations on graphs with 100,000 edges and varying amounts of nodes.

![Figure 4.2](image_url)  
**Figure 4.2**: Data from table 4.2 plotted on nodes and runtime using logarithmic scales on both axes.

In both table 4.2 and figure 4.2 it is shown that the CPU runtime corresponds directly to the increase of nodes in the graph. However, the runtime on GPU is very high at the starts, and decreases when adding more nodes to the graph. What is interesting in this data is that the runtime of the GPU implementation seems to be more correlated to the depth of the graph, rather than the amount of nodes present.
### Table 4.3: Comparing runtime between CPU and GPU implementations on graphs with 1,000,000 nodes and varying degrees of depth.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>Depth</th>
<th>Avg. Runtime CPU (ms)</th>
<th>Avg. Runtime GPU (ms)</th>
<th>Runtime ratio (CPU/GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^6$</td>
<td>$5 \times 10^5$</td>
<td>1</td>
<td>120</td>
<td>8.7</td>
<td>14</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>10</td>
<td>95</td>
<td>17</td>
<td>5.6</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^2$</td>
<td>124</td>
<td>66</td>
<td>1.9</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^3$</td>
<td>130</td>
<td>410</td>
<td>0.32</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^4$</td>
<td>88</td>
<td>2700</td>
<td>0.032</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^5$</td>
<td>78</td>
<td>28000</td>
<td>0.003</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>75</td>
<td>280000</td>
<td>0.0003</td>
</tr>
</tbody>
</table>

**Figure 4.3:** Data from table 4.3 plotted on depth and runtime using logarithmic scales on both axes.

Table 4.3 was generated by the C-program, keeping nodes constant and varying the depths of the graphs. Figure 4.3 clearly shows that the GPU runtime corresponds directly to the depth of the graph, while the CPU implementation barely takes it into consideration.
<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
<th>Depth</th>
<th>Avg. Runtime CPU (ms)</th>
<th>Avg. Runtime GPU (ms)</th>
<th>Runtime ratio (CPU/GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>262,111</td>
<td>1,234,877</td>
<td>63,791</td>
<td>13</td>
<td>6,600</td>
<td>0.002</td>
</tr>
<tr>
<td>Epinions</td>
<td>75,879</td>
<td>508,837</td>
<td>6,955</td>
<td>13</td>
<td>780</td>
<td>0.017</td>
</tr>
<tr>
<td>Facebook</td>
<td>4,039</td>
<td>88,234</td>
<td>346</td>
<td>0</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>Gnutella p2p network</td>
<td>62,586</td>
<td>147,892</td>
<td>5,469</td>
<td>10</td>
<td>3,400</td>
<td>0.0029</td>
</tr>
<tr>
<td>Google web graph</td>
<td>875,713</td>
<td>5,107,039</td>
<td>66,960</td>
<td>280</td>
<td>19,000</td>
<td>0.015</td>
</tr>
<tr>
<td>Wiki talk</td>
<td>2,394,385</td>
<td>5,021,410</td>
<td>15,579</td>
<td>550</td>
<td>14,000</td>
<td>0.038</td>
</tr>
</tbody>
</table>

**Table 4.4**: Comparing runtime between CPU and GPU implementations on graphs based on data collected from real world networks.

Table 4.4 consists of graphs based on real world data downloaded from the SNAP dataset. All graphs are quite deep, and the runtime for CPU is consistently much better than for GPU.
5. Discussion

5.1 Analysis of Result

In the following sections, the results from section 4 is analysed and discussed.

5.1.1 CPU Analysis

It is apparent from the data in tables 4.1 and 4.2 that the runtime of the CPU implementations seems to depend on both the amount of nodes and the amount of edges present in the graph. This is explained by the fact that each node and edge has to be processed exactly once sequentially by the CPU. The amount of nodes does however seem to have a greater impact on runtime than the amount of edges. It is currently unknown why this is the case, possibly it is because of the requirement to allocate memory for each node before adding it to the queue. These results are fairly close to the expectations based on section 2.3 which states that Kahn’s algorithm works in linear time with complexity \(O(|V|+|E|)\). Because of this, the depth of the graph has no major significance on the runtime of the CPU implementation. This is confirmed by table 4.3. There does seem to be a slight discrepancy in runtime which might be caused by cache access due to how nodes and edges are placed in memory.

5.1.2 GPU Analysis

The runtime of the GPU implementation seems unaffected by the increase of the amount of edges in the graph, shown in table 4.1. However when the depth of the graph radically increases, the runtime does the same. This also seems to hold true in table 4.2, where the GPU runtime seems correlated with the degree of depth in the graph rather than the amount of nodes. Figure 4.3 clearly shows that the GPU runtime is almost linearly dependent on the degree of depth in the graph.

5.1.2 CPU and GPU comparison

When comparing the ratio of the runtimes for the two implementations, the data in tables 4.1, 4.2 and 4.3 presents strong evidence that the GPU implementation performs far better than the CPU implementation on graphs with shallow depths. On graphs with a depth lower than 10, the GPU implementation is on average about 10 times faster than the CPU implementation. However as the depth of the graphs increases, so does the runtime of the GPU implementation.
As Figure 4.3 shows, the GPU implementation seems to outperform the CPU implementation on the graphs with a depth of up to about 250. With an increased depth the runtime of the CPU implementation is preferable to that of the GPU implementation. Table 4.3 shows that on a graph with 1,000,000 nodes, 1,000,000 edges and a depth of 1,000,000, the GPU implementation only is 0.0003 as fast as the CPU implementation. Or inversely, the CPU implementation is 3700 times faster than the GPU implementation.

When examining how both implementations perform on graphs based on real-world data, table 4.4 shows that the GPU implementation is only on average 0.012 times as fast as the CPU implementation (inversely the CPU implementation being faster 83 than the GPU).

Since the CPU runtime depend on both the amount of nodes and edges, one cannot draw the conclusion that there is a certain depth on graphs in general where the GPU implementation outperforms the CPU one. One can however conclude that the GPU implementation performs better than the CPU implementation on graphs with a fairly shallow depth in relation to the amount of nodes and edges.

One thing to keep in mind when viewing these results is that no parallel optimizations were conducted on the CPU implementation. As mentioned in section 2.5, studies have shown that performance gained by porting programs to the GPU may be skewed due to the fact that people rarely compare parallel CPU implementations to GPU ones. If one is solely interested in as good performance as possible, the results in this report should therefore be viewed with this in mind.

Although the performance of the GPU implementation are in many cases inferior to that of the CPU implementation, it might be beneficial to use it in some cases. If a topological sorting needs to be performed on a system with a high workload, performing calculations on the GPU will relieve the CPU from performing the topological sort. It can therefore help relieving the CPU, letting it do other calculations, which would result in a higher overall performance on the system.
5.2 Code Analysis

The fact that the depth of the graphs have such a strong effect on the runtime of the GPU implementation is not surprising when analysing the code of the GPU implementation. In order to ensure the list produced by the program is a topological ordering of the nodes in the graph supplied to the program, layers must be handled one at the time. If layers were not to be handled one at a time, nodes from layers of a higher order might be appended to the list containing the result before all of the nodes from lower order layers have been calculated. This entails that the GPU might have many idle threads during parts of execution. Since these threads will not process nodes from the next layer until the current layer has finished, a significant amount of processing power might go to waste.

To ensure that layers were in fact handled correctly, the decision was made to construct a queue for the GPU (described in section 3.3). Someone unfamiliar with GPGPU programming might at first glance think that it would be better to construct the queue in a way more similar to the way it is done in the CPU implementation. This however is not a good idea since the GPU implementation does not handle one node at a time, as the CPU implementation does. Nodes might be added to the queue in the wrong order if added as soon as a thread in the GPU encounters a node with an indegree of zero, as is done in the CPU implementation. This solution would therefore give an erroneous result. Instead all nodes within a layer that has an indegree of zero must be added to the queue, and no others. Then the queue must be emptied, and lastly it must be reset. These three steps must be repeated as many times as the graph is deep.

Naturally, launching the three kernels becomes more expensive the more times they have to be launched. A following consequence would also be that the enqueuer kernel takes up a higher percentage of time the more times the kernels are launched. This is due to the fact that the enqueuer kernel must look through every single node each time it is launched. The dequeuer kernel on the other hand must only look at as many nodes as there are in the queue each time it is launched. Both the enqueuer and the dequeuer kernels make use of atomic operations, which as noted in section 2.7 may affect performance negatively. These operations are however crucial to avoid race conditions that would arise without them. Another limiting factor is that the shared memory of the GPU not could be utilized in the GPU implementation due to the fact that nodes connected by an edge often reside far apart in memory, as mentioned in section 2.2.

The last row in table 4.3 presents a worst case scenario of these problems which occurs when the depth of the graph is equal to the amount of nodes in the graph. In this particular scenario the three kernels must be launched 1,000,000 times each. The enqueuer kernel must examine 1,000,000 nodes 1,000,000 times to find the one node
with an indegree of zero for each layer. Since the dequeeuer in practice only handles one node at a time in this scenario, only one thread will be active when emptying the queue. This results in bad parallelism, which leads to very bad performance when compared to the CPU implementation.

5.3 Future Work

A GPU with CUDA compute capability higher than 3.0 was not accessible during testing. GPUs with compute capability 5.2 or higher are able to launch kernels within kernels. Since the GPU implementation launches kernels from the host such a high number of times in some cases, better GPU benchmark results might be achievable on a GPU with higher compute capability.

Further optimizations of the code might exist as well. Both representation of data in memory and the algorithm accessing the memory itself can be constructed differently. It might be possible to construct an improved solution giving better performance.

There is also the possibility that data generated is not representable of data used in real systems. It is difficult to draw a conclusion determining where and when the GPU implementation actually performs better in real world problems without DAGs coming from real world systems. Benchmarking such a dataset would be very beneficial in drawing conclusions on usefulness in real world systems.
6. Conclusion

The aim of this study was to investigate the performance of a naive parallel implementation of Kahn’s topological sorting algorithm on a GPU compared to the sequential CPU implementation. This study has identified that it is generally less effective to implement the algorithm on the GPU, although our findings indicate that for large graphs with shallow depths there are performance gains to be had. The study also shows that depths are of great importance in graph algorithms where layers are computed in a sequential manner. More research is required to determine the efficacy of such implementations, especially using newer techniques such as dynamic parallelism in new versions of CUDA. Notwithstanding these limitations, the study suggests that the GPU is a viable choice for relieving the CPU when performing calculations on large, shallow graphs.
7. References


GPU Pseudo Code Appendix

**Pseudo code for Enqueuer GPU kernel**

I ← Indegree Array  
Q ← Queue Array  
qs ← Queue Size  
nc ← Node count, size for arrays  
Calculate node number nn from Thread and Block  
If indegree for I[nn] is 0 and nn smaller than nc:  
  Set indegree I[nn] to -1  
  Add nn to next available index in Q

**Pseudo code for Dequeuer GPU kernel**

N ← Node Array  
Q ← Queue Array  
I ← Indegree Array  
R ← Result Array  
E ← Edge Array  
qs ← Queue Size  
nc ← Node Count  
Calculate node number nn from Thread and Block  
If nn is less than qs:  
  Add node number Q[nn] to R[rc], adds result to result Array  
  Increment rc by 1  
  For each edge offset k between N[Q[nn]] and N[Q[nn] + 1]:  
    Subtract 1 from I[E[k]]

**Pseudo code for Reset Queue GPU kernel**

qs ← Queue Size  
Set qs to 0

**Pseudo code for Outside CPU Loop**

nc ← Node Count  
Set Thread size ts to maximum value for GPU  
Set Block size bs to nc/ts + 1;  
Loop:  
  Launch Enqueuer with thread size ts and block size bs  
  Copy queue size qs from Device to Host  
  if qs is 0:  
    Break loop  
  Launch Dequeuer with thread size ts and block size qs/ts + 1  
Launch Reset Queue