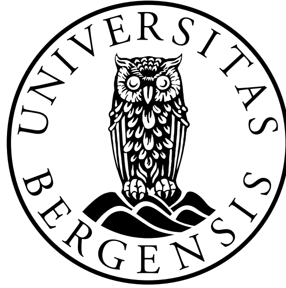


UNIVERSITY OF BERGEN

DEPARTMENT OF INFORMATICS

ALGORITHMS

Computing Connected Components on Multiple GPUs

Student:
Yngve Hellås

Supervisor:
Professor Fredrik Manne

Master Thesis
November 2018

Acknowledgement

First, I would like to thank my supervisor Fredrik Manne for supervising me and giving guidance throughout this thesis. Additionally, I would like to thank my family and friends, especially my parents and Mathias for the constant support. Thanks to my fellow algorithm master student for support and making the study hall a great place to be.

Contents

1 Introduction **5**

2 Background **6**
2.1 Parallel programming . . . 6
2.1.1 Memory . . . 7
2.2 CUDA - GPU parallel programming . . . 8
2.3 Connected components . . . 10
2.3.1 Solutions for connected components . . . 10
2.3.2 Connect high to low . . . 11
2.4 Rem's algorithm . . . 11
2.4.1 Sequential . . . 11
2.4.2 Parallel . . . 12
2.5 Test computers . . . 14

3 Gunrock **16**
3.1 What Gunrcok offers . . . 16
3.2 Experience with Gunrock . . . 16
3.2.1 Connected components . . . 17
3.2.2 Move to Groute . . . 17

4 Groute **18**
4.1 What Groute offers . . . 18
4.2 Connected Components in Groute . . . 19
4.2.1 Hook-non-atomic . . . 21
4.2.2 Hook-atomic . . . 24
4.2.3 Compress . . . 27
4.2.4 Merge . . . 29
4.2.5 Final-compress . . . 30
4.3 Implementation of Rem in Groute . . . 30

4.3.1 Adjusting Rem's algorithm for implementation 31
4.4 Experience with Groute 31

5 CUDA Implementation 33
5.1 Application structure 33
5.1.1 Changes for implementation 35
5.2 Communication - data transfer 37
5.2.1 Distributing graph 37
5.2.2 Merging solutions 37
5.3 Handling multiple devices 38
5.4 Experience from multi-GPU programming 39

6 Experiments 40
6.1 Graphs 40
6.2 Test measurements 43
6.3 Groute experiemnts 44
6.3.1 Hook-non-atomic 44
6.3.2 Rem and compress operations 46
6.3.3 Run time perfromance Hook vs Rem 46
6.3.4 Strong scaling 53
6.3.5 Weak scaling 54
6.3.6 Speed-up 54
6.3.7 Expanded graphs 55
6.4 CUDA implementation experiments 56
6.4.1 Run time performance Hook vs Rem 57
6.4.2 Compare run time between CUDA and Groute implementations 58
6.4.3 Strong scaling 58
6.4.4 Weak scaling 59
6.4.5 Speedup 60
6.5 Distribution bottleneck Google Cloud 61

7 Conclusion 63
7.1 Experience 63
7.2 Rem and findings 63
7.3 Further work 64

Bibliography 65

Chapter 1

Introduction

As GPU programming is improving along with the performance of GPUs, is GPU programming getting more interesting by the day. An expansion to parallel programming on one GPU is parallel programming using multiple GPUs. For this thesis we focus on CONNECTED COMPONENTS algorithms as we look into multiple GPU programming.

In this thesis we give a short introduction to GPU programming through the CUDA programming language and connected components, along with other terms that are useful for the rest of the thesis. We look at two systems that aid in multiple GPU programming, the CUDA library Gunrock and the runtime environment Groute. We start with Gunrock and why we switched to Groute, before we describe Groute. As a comparison we create our own implementation in CUDA for multiple GPUs, without any supporting systems. We perform experiments on Groute to get an idea of its performance. We present a fast CONNECTED COMPONENTS algorithm that we believe to be faster than algorithms that currently are implemented as standard on Gunrock and Groute, this algorithm is implemented in Groute and we compare the performance between the already implemented CONNECTED COMPONENTS and this faster algorithm.

Chapter 2

Background

In this chapter are we describe background information, terms and techniques used in this thesis. We present terms and functions used in parallel programming and terms that are specific with CUDA parallel programming. We describe other topics such as Connected components and the computers used for testing.

2.1 Parallel programming

With parallel programming do we want to perform multiple task at the same time to increase the run time performance.

Asynchronous

When using asynchronous operations processors and threads can work at their own pace [13]. In CUDA does a synchronization operation block the computation stream, this prevents other operation from proceeding, while an asynchronous operation allow other operation to proceed[1]. Kernel calls in CUDA are all asynchronous [13]. The host and devices each have their own memory, distributed memory, thus there is need for communication when sending information from host to device, device to host or device to device.

Bulk-synchronous

When a program uses a bulk-synchronous approach processors are working in parallel until they reach a synchronization step. When a processor reaches this step will the processor wait for all the other processors to reach the same step. Synchronization can be used to prepare the processors for communication with other

processors or as a scheduling for a task where it is necessary for the processors to be synchronized.

Host and device

Two terms used for parallel programming with CUDA is host and device, host refers to the CPU (central processing unit) and its memory, device refers to the GPU (graphical processing unit) and its memory. The CUDA compiler separate the code for each part before the host compiles with C and the device compiles with CUDA C [1]. Individually do host and device have shared memory, but when we use CUDA is host and device part of a distributed memory system as the memory for the host is not directly accessible by the device and vice versa. The host is the general-purpose processing unit, handling more complex task and usually have faster calculating speed. While the device is getting more general purpose is it still better at performing many simpler tasks. Though the computing speed of a device is slower than the host is the device faster on big workloads because of its parallel work structure.

2.1.1 Memory

Memory is a collection of location on a computer or device. The locations are capable of storing data and instructions[13].

Shared-memory

We are describing two memory-systems that is used in this thesis, the first is shared-memory. In shared-memory programs can all threads read and write to the same variables, these variables are called shared. There are also private variables, private variables can usually only be accessed by one thread. By writing and reading shared variables can threads communicate with each other and share information. There is no need for explicit communication between the threads [13]. Shared-memory is the normal system for CPU programming, where the memory is the RAM.

Distributed-memory

The second memory-system is distributed-memory. In distributed-memory is the memory private for each core, notice not thread. Usually is distributed-memory used on hardware where cores have their own memory. With private memory are there no shared variables as there was in shared-memory, this requires explicit

communication between the cores to share variables and information[13]. With GPU programs is distributed-memory used. The GPU and the CPU each have their own memory and need to communicate to send variables from the CPU to the GPU and back again.

Race condition

When we need to handle multiple threads or processes can a problem arise where these threads try to change the same piece of data at the same time. This is called a race condition. The result in this races depends on which thread or core that wins the race[13]. Let's use an example. Take a program where thread 1 and thread 2 tires to write their ID to variable A at the same time, and then outputs value of A. If this program would run several times could the output varied between 1 and 2. Handling race conditions can be solved by rewriting a program where it avoids the possibility of a race condition, or use a function as atomic operations, described next.

Atomic operations

To solve the race condition problem can we use atomic operations. Atomic operations prevent multiple threads interfering with each other when accessing the same data. This is done by locking the piece of data that is accessed by multiple threads when a thread is performing an operation on it[1].

There are three different types of atomic operations, arithmetic functions, bitwise functions and swap functions. The swap function is used in connected components program in chapter 2.4 and 4. The swap function used is atomicCAS, CAS stands for compare and swap. AtomicCAS conditionally swaps a value if the stored value matches the specified value [1].

2.2 CUDA - GPU parallel programming

In this section we give a brief introduction to a GPU programming language. To gain access to the possibility of parallel programming on a GPU is CUDA the favoured language for many. When using a GPU in parallel programming is the CPU still an important component, as it sends the instructions and data to the GPU, where instructions can be done on a higher number of simultaneous executions. As the CPU and GPU each have their own memory, a distributed memory system, is there a need for communication when sharing data. If multiple

GPUs are used, then there is need for communication between them if they need to share information. Communication between devices can be done directly between GPUs if the hardware is structured in such a way that allow this. If two GPUs in as system can't communicate directly is the communication done through the CPU. Sending and receiving data in GPU programming is among the more time-consuming task as the speed at which data can be transferred between CPU and GPU is much slower than a normal memory access for each of the CPU or GPU. For this reason, problems with a need for more computation steps compared to the data used often gaining more from GPU programming.

Memory

There are multiple possibilities for where to store data in the memory of a GPU, in CUDA there are six types of memory on a GPU. Each type of memory has different sets of attributes, as with memory on a host is there a trade of with speed and size. There small and fast memory as register and local memory which are accessible by one thread and is only active as long as the thread is active. For threads in a block there is a shared memory among these threads, the lifetime for this memory is as long as the block is active. Constant memory is accessible for both host and the GPU but is read only for kernels. The global memory is the largest highest latency and most commonly used memory. [1]

Threads in CUDA

Threads in CUDA are organized in blocks, a block can hold 1024 threads and these threads can be stored in one, two and three dimensions. Blocks are stored in grids, as for threads can block be stored in one, two and three dimensions. The number of blocks in a grid dimension is limited to 65535 blocks. When threads are sent to a streaming multiprocessor are the threads stored in warps. A warp can hold up to 32 threads, threads in a warp compute in lock steps. All threads in a warp must execute the same instructions, if some threads has instructions to execute that other threads in the warp don't then these threads have to wait while the threads with instructions execute. This is called warp diverges [1]. This can happen with if statements, if some of the threads have to do the true condition while the other have to do the false condition. Warps gets distributed to streaming multiprocessors, which are groups of CUDA cores [15]. The streaming multiprocessors arrange and prepare warps for execution such that the CUDA cores are constantly working as long as there is work for them to do.

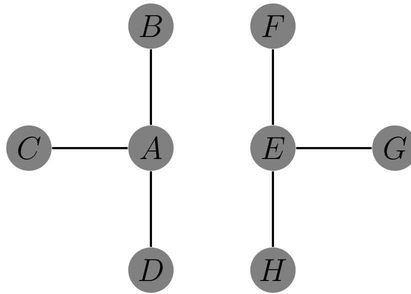


Figure 2.1: Two connected components

Kernel

A kernel function is a function that is executed on the device. In the kernel code are the instruction for a single thread, which is to be performed in parallel with other threads defined in the kernel call [1]. In kernel calls we define the number of threads being used with grids and blocks, the input for a kernel call looks like a function call in C, but a kernel call doesn't have a return value. To return result from a kernel call must there be a copy of data from the device to the host.

2.3 Connected components

An undirected graph is connected if there is a path between every node, for any node can we move along the edges of the graph to all the other nodes. If there are parts of the graph that is connected is these called connected components [9]. In Figure 2.1 is a graph with two connected components, the left component contains A, B, C, D the right component contains E, F, G, H. The goal for CONNECTED COMPONENT is to mark nodes with the name of the component it is connected to. When all nodes are marked can we count the number of connected components by counting the number of different components names.

2.3.1 Solutions for connected components

There multiple ways to solve CONNECTED COMPONENT, we can use searching algorithms as BREATH FIRST SEARCH and DEPTH FIRST SEARCH marking node as the algorithm moves through them. Another solution is to connect nodes and add them to a tree structure where the ID of the top node in the tree structure is the name of that component. This solution is a UNION-FIND algorithm.

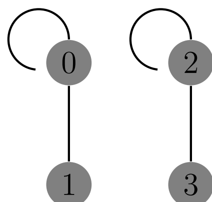


Figure 2.2:

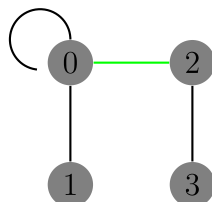


Figure 2.3:

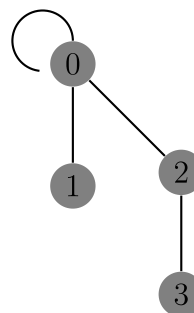


Figure 2.4:

Figure 2.5: Connecting nodes 0 and 2 using their node ID

2.3.2 Connect high to low

A technique that algorithms that connects nodes into components can use is to use the integer values node IDs to decide which node to be set as the parent of the other, instead of choosing this randomly. When we get two nodes connected by an edge, we set node with the lowest value node ID as the parent of the node with the higher valued ID. Figure 2.5 shows this technique, here there are two components, we receive an edge(0, 2), as node 0 is the lower valued node is it set as the parent of node 2. This technique can be use by setting low to high as well, its important that this is consistent in the whole program.

2.4 Rem's algorithm

In this thesis we want to show that Rem is a fast parallel algorithm solving CONNECTED COMPONENTS. In this section we introduce the sequential and parallel version of Rem's algorithm. Rem's algorithm use the technique described in Section 2.3.2 as it connects nodes and components into connected components.

2.4.1 Sequential

Algorithm 1 show a sequential version Rem's algorithm from "Experiments on union-find algorithms for the disjoint-set data structure" [14]. The algorithm takes an edge(x, y) as input, x and y are integer value representing the node ID. In line 1 are these stored in r_{x} and r_{y} respectively. In line 2 a while loop starts and loops as long as the parent of r_{x} and r_{y} are not the same value. In line 3 the parents are

Algorithm 1 Rem(x, y)

```
1: r_x ← x, r_y ← y
2: while p(r_x) ≠ p(r_y) do
3:     if p(r_x) < p(r_y) then
4:         if r_x = p(r_x) then
5:             p(r_x) ← p(r_y), break
6:         end if
7:         z ← r_x, p(r_x) ← p(r_y), r_x ← p(z)
8:     else
9:         if r_y = p(r_y) then
10:            p(r_y) ← p(r_x), break
11:        end if
12:        z ← r_y, p(r_y) ← p(r_x), r_y ← p(z)
13:    end if
14: end while
```

compared, if p(r_x) is lower than p(r_y) then r_x is checked to see if it is a root node in line 4. If it is a root node p(r_y) is set to the parent of p(r_x) and breaks out of the while loop. If r_x isn't a root node are the parent values updated to scale the parent tree in line 7. If p(r_x) was bigger than p(r_y) we skip to line 9. Here r_x is checked if it's a root node. If it is a root node is p(r_x) is set as the parent to p(r_y) and the the algorithm breaks out of the while loop to receive a new edge, if is not a root node the parent are values updated and the algorithm goes back to line 2.

2.4.2 Parallel

Algorithm 2 show the parallel version of Rem's algorithm, we received the algorithm from our supervisor Fredrik Manne [2]. This algorithm is made for running in parallel on a GPU. As threads on a GPU work in lock step is the goal for optimizing code for GPU to let threads have work on all steps and avoid steps where threads are idle, this is called warp divergence. The important difference between Algorithm Algorithm 1 and Algorithm 2 is that Algorithm 2 swaps the values when the parent of x is bigger than the parent of y, instead of using two if conditions. The swap is used such that the y values always represent the bigger value.

The input for the Algorithm Algorithm 2 is an edge(x, y). As there is possibility that values are swapped between x variables and y variable are both the node and the parent ID stored in variables, this also reduces the calls to memory. Lines 1 through 4 sets the node and parent values. In line 5 a while loop start, and loops as

long as the parents of x and y are not the same. In line the parents are compared, if pr_x is bigger all values for x and y swap, this happens in line 7 to 12. Both the node value and parent value are swapped such that the parent value is correct. If pr_x is smaller the swap is not necessary, as parent of y is bigger than the parent of x. In line 14 an atomic compare and swap operation is used, this locks the current parent value of r_y. If $p(r_y)$ is equal pr_y, then pr_x is set as the parent to r_y. The result from the atomic operation is stored in result. If the compare failed the result is set to the new parent that r_y, as the compare and swap failed since another thread changed the parent of r_y. In line 15 the result from the atomic operation is compared with the locally stored parent value, if they are different the local parent value is updated, and the algorithm goes to line 5 again. If not, the parent values are updated, where r_y is set to the local parent o r_y, and we find this nodes parent. This update compresses the parent trees created by the algorithm.

Algorithm 2 RemParallel(x, y)

1: $r_x = x$
2: $r_y = y$
3: $pr_x = p(r_x)$
4: $pr_y = p(r_y)$
5: **while** $pr_x \neq pr_y$ **do**
6:     **if** $pr_x > pr_y$ **then**
7:         $tmp = pr_x$
8:         $pr_x = pr_y$
9:         $pr_y = tmp$
10:         $tmp = r_x$
11:         $r_x = r_y$
12:         $r_y = tmp$
13:     **end if**
14:     result = atomicCAS($p(r_y)$, pr_y, pr_x)
15:     **if** result $\neq pr_y$ **then**
16:         pr_y = result
17:         continue
18:     **end if**
19:     $r_y = pr_y$;
20:     $pr_y = p(r_y)$
21: **end while**

	Tesla K40	**Tesla K80**
Memory size	12 GB GDDR5	24 GB GDDR5
Memory Bandwidth	288 GB/s	480 GB/s
Memory clock	3.0 GHz	2.5 GHz
CUDA cores	2880	4992
Processor core clock	745 MHz	560 MHz
Processor core clock		562-875 MHz Boosted

Table 2.1: Specification for K40 and K80 graphics processors

2.5 Test computers

We use two computers for test in this thesis. The first is Lyng, a shared compute server at the University of Bergen. Lyng has two Xeon E5-2699 v3 CPU, 2,30 GHz with 252 GB ram, connected with one Nvidia Tesla K40m graphic processor. The second computer is a Google Cloud server, which is a rent server service [3]. The Google Cloud server runs a virtual CPU with 60 GB ram, connected with eight Nvidia Tesla K80 graphic processors. Nvidia Tesla K80 is the better than Nvidia Tesla K40m, K80 has more memory, faster transfer of data and more CUDA cores. Information for each graphic processor is shown in Table 2.1.

Figure 2.6 show the GPU structure for the Google Cloud computer. There are two groups of four GPU's, where all GPU's in a group are connected to each other. GPU's that are connected can send and receive data from each other without going through the CPU. If a GPU from one group communicate with a GPU in the other group, then the communication go from the GPU through the CPU and then to the other GPU. All GPU's are connected to the CPU.

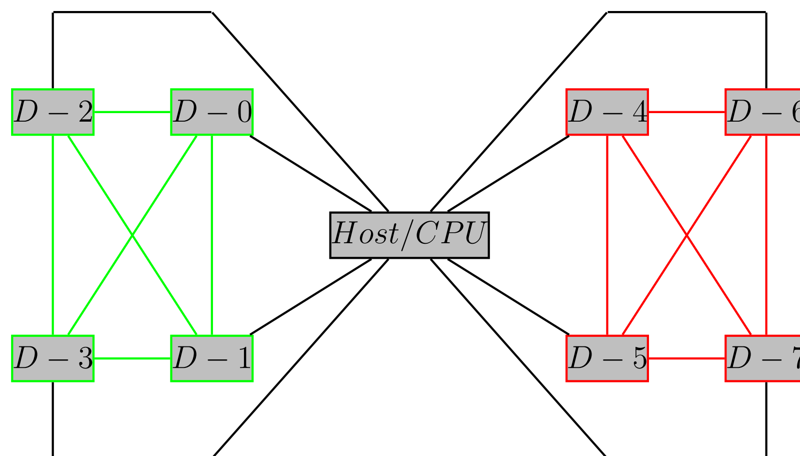


Figure 2.6: GPU structure on Google Cloud server, two groups of devices

Chapter 3

Gunrock

In this chapter we look at the GPU library Gunrock. Gunrock is a high-performance graph processing library for GPU programming using a bulk-synchronous programming approach. Gunrock achieves a balance between performance and expressiveness with the approach to GPU programming [16].

Gunrock was the first library or environment considered for this thesis. Gunrcok caught our attention with possibilities of high scalability on multiple GPU's for graph primitives. We look at what Gunrock is and our experience using it. Code for Gunrock can be downloaded from Github [5], there is also a website [6] where instructions, documentation and code can be found. This includes how to install and use the library.

3.1 What Gunrcok offers

Gunrock present a data-centric abstraction to allow development of graph primitives at a high level of abstraction, while maintaining high performance [16]. By using Gunrock we can use their GPU specific optimization strategies for memory efficiency, load balancing and workload management [16].

3.2 Experience with Gunrock

Our intentions with Gunrock were a two-step experience with the library. First we wanted to look at and replace the CONNECTED COMPONENTS implementation. In the second step we wanted to implement another algorithm, without using any pre-existing algorithm to see how easy or hard implementing in Gunrock would be.

	CC(ms)	
Graph	Gunrock	Groute
USA	335.65 (1)	**15.11 (5)**
OSM-euro-k	-	**160.96 (4)**
soc-LiveJournal1	110.05 (1)	**14.19 (2)**
twitter	-	**384.13 (8)**
kron21.syn	-	**13.86 (8)**

Table 3.1: Best running time comparison between Gunrock and Groute, number of GPU for best performance in parentheses [8].

3.2.1 Connected components

Our goal in the first step of implementation was to find the code for Connected components and replace it with our Rem algorithm for Connected components, presented in Section 2.4. We do this for two reasons, first is getting acquainted with how Gunrock code is structured around a graph and achieve a better understanding for how we would implement a new algorithm. The second reason is to test how Rem's algorithm perform in this library, how well it scale on multiple devices and how well do it performed compared to the original implementation.

While working on the first step finding and replacing the Connected components code, there were no website for Gunrock, only the code and install instruction on Github. We looked through the code following a run step for step. At this point we could not determine where they called a function that looked like Connected components that we could replace or modify to implement Rem's algorithm. Gunrock was installed and tested on Lyng, see Section 2.5, the Connected components implementation worked, but we could not find the necessary part to continue.

3.2.2 Move to Groute

As we were stuck in the process of implementing Rem's algorithm we found another runtime environment for multiple-GPU programming. The new runtime environment is Groute, see Chapter 4. In their paper, Groute: An asynchronous multi-GPU programming model for irregular computations [8], they show that Groute and Gunrock varied between who had the better performance on different algorithms and graphs. Crucially Groute was faster on all Connected components comparisons, which is where our interest are. Table 3.1 shows the performance comparison between Gunrock and Groute from this paper for Connected components.

Chapter 4

Groute

In this chapter we introduce the runtime environment Groute and take a closer look at its CONNECTED COMPONENTS implementation. We also show an implementation of the Rem algorithm from Section 2.4 in Groute to compare these.

Groute is an asynchronous programming model and runtime environment for multi-GPU programming created by Tal Ben-Nun, Michael Sutton, Sreepathi Pai and Keshav Pingali. The motivation for creating Groute was to give a more optimal runtime environment alternative for irregular algorithms which may perform better with an asynchronous programming approach as opposed to bulk-synchronous programming [8]. The code for the Groute runtime environment is available on Github [4]. There is no documentation of the Groute code or environment, therefore the algorithms mentioned below are extracted from the code on Github [4].

4.1 What Groute offers

The Groute runtime environment offer us a way to handle communication and data transfer for multi-GPU programming. Transferring data is a time consuming part of GPU programming, having a environment that speed this up can increase performance. Groute environment consist of three layers, a low-level management of topology and inter-GPU communication, communication constructs that optimize memory transfer paths and a distributed work list [8].

Further on are we focusing on Pending segments, which are called input_segment and merge_segment in the CONNECTED COMPONENTS code we show in Section 4.2. Pending segment represent segments that are being received, the host divides input into Segments [8]. These segments are sent to available devices. In Section 4.2 is input_segment edges from the input graph, while merge_segment are edges

received from other devices when a solution is collected.

4.2 Connected Components in Groute

Groute offers implementation of Breath first search, Page rank, Predicted-Based Filtering, Single source shortest path and Connected components. We are in this section describing Groute's implementation of Connected components.

As described in Section 2.3 the goal for Connected components is to mark nodes by which component they are in, when components are marked can they be counted, to get the number of connected components in a graph. The way Groute solves this is with a version of Union-find, slightly modified compared to the description in Section 2.3. These enables the code to handle multiple devices. When using multiple devices the graph is first distributed between the devices. Each device then calculates a local solution before all the solutions are merged together to one final solution. Every device has all the nodes in the graph, while the edges are distributed between the devices such that an edge only exists on one device. A device then computes connected components from the edges it has received. As the devices cannot see each others memory, the solution from each device is only a part of the final solution for the whole graph. The algorithm must therefore collect the solution from each device and merge these together to achieve the final solution.

We describe the Groute's connected components implementation in Algorithm 3. This algorithm is separated into two while loops. In the first while loop (lines 1 to 8) each device computes components on its assigned edges. This is done using a modified Union-find algorithm. The second loop collects and merges results from each device into the final solution. If this is not done, we would end up with an incomplete solution on each device. The algorithm ends with the Final-compress algorithm, which compress the parent structure which is needed for Groute to recognize the solution.

To understand how Algorithm 3 runs we start with the first while loop, the modified Union find. The while loop runs as long as there are input segments containing edges from the input graph. There are two main functions happening in the while loop, hook and compress. In lines 3 and 6 the algorithm performs Hook operations that connects nodes by setting one node as the parent f another. Both Hook functions use the technique explained in Section 2.3.2, where the integer values of the nodes are used to set a lower numbered node as the parent of a higher numbered one. In this was the nodes are connected into tree structures reflecting

Algorithm 3 Connected components in Groute

```
1: while Pending input_segment do
2:     for Number of Hook-non-atomic iterations do
3:         Hook-non-atomic
4:         Compress
5:     end for
6:     Hook-atomic
7:     Compress
8: end while
9: while Pending merge_segment do
10:    Merge
11: end while
12: Final-compress
```

which nodes are in the same connected component. Hook-non-atomic and Hook-atomic are explained in Section 4.2.1 and Section 4.2.2 respectively. After each call to a Hook function there is a compression operation (lines 4 and 7) to reduce the height of the trees given by the parent pointers. This is described in Section 4.2.3.

After the first loop is completed Algorithm 3 continues to collect the solution in the second while loop. This is done by processing merge segments, which contains edge lists specifying the parent trees from the first while loop. This is only needed when we are using more than one device to run Algorithm 3. The merging is done in line 10 by performing Hook-atomic and is described in Section 4.2.4. As these operations changes the tree structure we need to compress the final tree. This is done in the Final-compress algorithm, explained in Section 4.2.5.

Parallel execution

Now that we have an overall understanding of what Algorithm 3 does, we can explain how this is done in parallel. There are two levels of parallelism that are running at the same time when Algorithm 3 is running on more than one device. The first level is the devices that runs concurrently. This level is set up before Algorithm 3 starts using the communications tool mentioned in Section 4.1. Each device then runs Algorithm 3 separately. There is no direct synchronization between the devices as there would be in in a bulk-synchronized program. Each device runs the first loop as long as there are input segments to work on. When a device runs out of input segments it continues to the second loop. There is no synchronization between the devices when they move from the first to the second

loop. As the workload is distributed continuously to devices the workload is divided fairly evenly. Thus, the devices are expected to finish the first loop at about the same time. Each device thus participates in the second loop until there no more merge segments left to process. Merge segments are edge list from other devices parent structures. Devices have a partner device that either they send their parent structure edge list to or receive a one from. A device that have sent this edge list is done, devices that received an edge list merge this with their parent structure. Devices that are still active finds a new partner device, this continuous until one is left with the final solution from all devices.

The second level of parallelism is happening on each device. Each device runs Algorithm 3 independently. Edges are received in bulk from the host during the execution of the algorithm. On the device operations in lines 3 to 7 and 10 in Algorithm 3 are run in parallel as kernel calls. The number of threads created for a kernel call varies. Each call to one of the Hook functions takes a list of edges and assigns one thread to each edge. For the compress calls each thread receive a node, so the number of threads is the same as the number of nodes in the input graph.

4.2.1 Hook-non-atomic

The Hook-non-atomic operation is given in Algorithm 4. This is a fast algorithm for creating a connection between two nodes. The trade-off for this speed is that it may overwrite existing connections when it runs in parallel. We describe later in this section how this might occur. Hook-non-atomic is called from inside a for loop in Algorithm 3 (lines 2 to 5) along with a compress operation. The purpose of this is to run Hook-non-atomic a fixed number of iterations. If the for loop runs for as many iterations as needed for there to be no more changes done by the Hook-non-atomic and Compress algorithms we would get a correct solution for current edges. Groute does not. However, the purpose of this for loop is not to complete an exact solution but rather to enhance performance. At the end of this section we show how the number of iterations impacts performance. Hook-non-atomic has the potential to be substantially faster than Hook-atomic but might not return a correct solution. We can exploit this while ensuring a correct solution by always having a final application of Hook-atomic that fixes any inconsistencies left by Hook-non-atomic.

Algorithm 4 take pointers to two lists as input, the parent list and edge list, both lists are shared between all threads on a device as they are stored in the shared memory on the device. The parent list i accessed with write and read

Algorithm 4 Hook-non-atomic(parents, edges)

```
1: Edge e = edges.getEdge(threadID)
2: e.u = parents.read(e.u)
3: e.v = parents.read(e.v)
4: high = e.u > e.v ? e.u : e.v
5: low = e.u + e.v - high
6: parents.write(high, low)
```

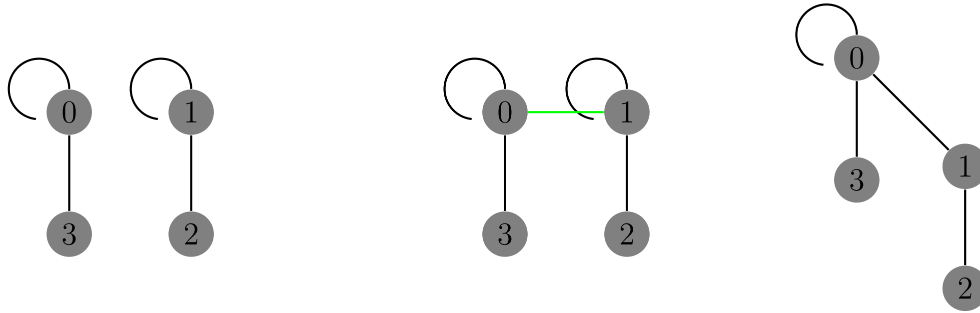


Figure 4.1: Hook-non-atomic connecting two nodes.

operations, lines 2, 3 and 6. In line 1 the thread finds an edge using its thread ID in the edge list, using thread ID no threads work on the same edge. Algorithm 4 uses the connecting high to low technique described Section 2.3.2. The edge from line 1 gives two nodes connected by this edge (u, v). Parents of both nodes u and v are set in lines 2 and 3. The parents is set such that the lower value becomes the parent of the higher valued one. In Algorithm 4 the function parents.read(e.u) returns the parent of u, while parents.write(high, low) sets high to point to low, high and low are parent of either u or v. An example is shown in Figure 4.1. In the leftmost graph there are two components, node 0 is the parent of node 3 and node 1 is the parent of node 2. Both node 0 and 1 are root nodes. Hook-non-atomic gets an edge (2, 3). As the algorithm goes to lines 2 and 3 it fetches their parents, node 1 and 0 respectively. Lines 5 and 6 outputs that node 1 should connect to node 0 as $0 < 1$. This is indicated by the green arrow in the middle graph. In line 6 node 0 is set as the parent of node 1, leaving us with the graph to the right in Figure 4.1.

We stated that a problem with Hook-non-atomic is that it can overwrite existing connections when executing in parallel. This could cause an already connected component to become disconnected. The overwrite problem can occur in line 6 of Algorithm 4, and is due to a race condition when two or more threads tries to

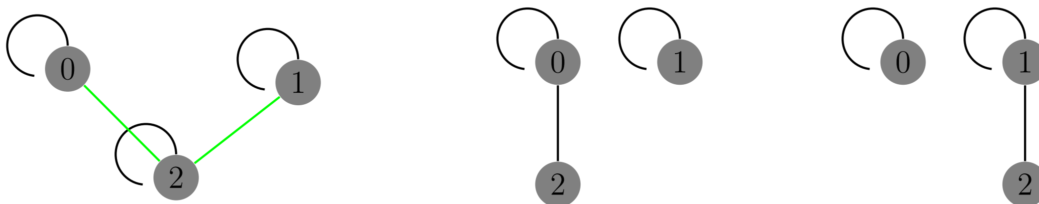


Figure 4.2: Hook-non-atomic faling in conencting nodes due to a race condition.

set different parent values for the same node simultaneously. Then only the value from the last thread to write will be kept, while all other operations will be lost. This may result in two nodes along with their current components to not become connected, thus leading to an incorrect result. Figure 4.2 shows the steps when a race condition occurs and gives the wrong result. To the left there are three nodes that are roots. There are two edges, marked in green, that are input to two different threads both running Algorithm 4. The first thread gets the edge (2, 0) and the second thread gets the edge (2, 1). Both threads then simultaneously find which node is high and low. The first thread then sets the parent value for node 2 to 0, shown in the middle graph in Figure 4.2. The second thread does not see this and thus sets the parent value of node 2 to 1. Leaving two components, shown in the right graph in Figure 4.2, which is clearly incorrect. One way to handle this race condition is using an atomic operation when setting the parent pointer as is done in the Hook-atomic algorithm, see Section 4.2.2.

A second problem Hook-non-atomic can encounter is separating components if the tree height of the parent structure gets too high. When Hook-non-atomic is connecting nodes it only looks at the parent nodes of the nodes it tries to connect. This yields the right solution if the parent nodes are root nodes. If they are not, then there might occur a separation from the nodes above the parent in the tree structure. Figure 4.3 shows an example with two components and where we try to connect the bottom node (node 3) with a new component (node 1). In Hook-non-atomic node 1 and node 2 are compared as they are the current parents of 1 and 3. Node 2 is then connected to node 1 since node 1 has a lower ID value than node 2. This leaves node 0 separated from node 2 and node 3. As a result, there are still two components.

Since Hook-non-atomic does not consider race conditions and can separate components when the tree height is more than two, the result from this algorithm can be incorrect when it runs in parallel. It is important to note that the connections made by Hook-non-atomic are still correct, the problem is that we might not get all the required connections. The main advantage of Hook-non-atomic is that an

Figure 4.3: Hook-non-atomic separating two nodes from a component

ensuring application of Hook-atomic is likely to run faster as it improves the parent structure before applying Hook-atomic. It is possible to perform multiple iterations of Hook-non-atomic on all the edges to get even closer to a correct result or we can run the Algorithm 3 with no iteration of Hook-non-atomic and only use Hook-atomic.

4.2.2 Hook-atomic

Algorithm 5 shows the Hook-atomic function. This ensures that all connections between nodes and components given by the input graph are included in the final solution of Algorithm 3. Hook-non-atomic is simple and fast, but as explained in Section 4 the solution can be incomplete. With Hook-atomic we get a complete solution, but at the cost of a few more steps and also an atomic operation that makes it slower. With Algorithm 5 we get two new features compared to Hook-non-atomic. The first is the atomic operation in line 8, see Section 2.1.1 for more on atomic operations. Adding an atomic operation to the step where the parent value of a node is changed ensures that two or more threads will not overwrite each other unintentionally. If the atomic write fails because the parent value has been changed or the high parent node is not a root node, then a thread will continue executing. This also handles the separation of components problem in Hook-non-atomic. The second feature is a while loop running from lines 5 to 15. This loop goes on as long as the parents that are compared are not equal or until the atomic write is successful, or another thread has updated low as the parent of high. The purpose of this loop is to ensure that a write to parents happens if the nodes in question are not already in the same component. If the atomic write is not successful and low is not set as the parent of high by another thread, then both parents are updated. This update gets the new parent for both nodes, at least one of the nodes has a

new parent as both parents can't be the same as before the update. If they are the same both are roots, and the atomicCAS would be successful and we would exit the loop.

Algorithm 5 Hook-atomic(parents, edges)

```
1: Edge e = edges.getEdge(threadID)
2: if e.u ≠ e.v then
3:     pu = parents.read(e.u)
4:     pv = parents.read(e.v)
5:     while pu ≠ pv do
6:         high = pu > pv ? pu : pv
7:         low = pu + pv - high
8:         prev = parents.write_atomicCAS(high, high, low)
9:         if prev == high || prev == low then
10:            break;
11:        end if
12:        pu = parents.read(prev)
13:        pv = parents.read(low)
14:    end while
15: end if
```

The progress through Algorithm 5 starts by getting an edge as input from the input segment and compare the nodes connected by this thread, if the nodes have the same value then its a self-loop and its not necessary to connect. When this runs in parallel on a device one thread will handle one edge and perform the steps in the algorithm in parallel with other threads. There is no synchronization before all threads have gone through all the steps using their edge. From the input edge we get the two nodes we are connecting, e.u and e.v. We take the parent of both nodes and go into the loop when these parents are not equal. If the parents are equal, then the nodes are already in the same component and no further action is needed. If the parents are not equal we continue to find which parent is high and low, as was done in Section 2.3.2. In line 8 we have the atomic write operation, if another thread is accessing the parent value that this thread is trying to access, then it must wait for the parent value to be available. If the write is successful, then the loop breaks as the nodes are now connected. When the write is unsuccessful then the result from the atomic operation is controlled, if it is equal to either high or low the thread is done as another thread have connected the nodes.If the result is not equal high or low then the loop continues and the algorithm updates the

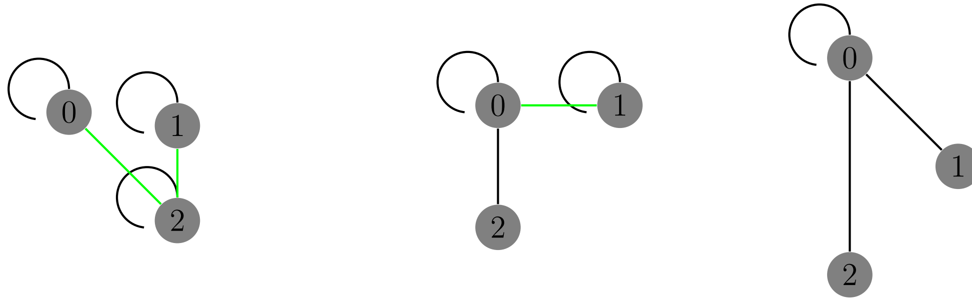


Figure 4.4: Hook-atomic correctly connects node even with a race condition present.

parent values in lines 12 and 13. In line 12 and 13 we advance up the parent tree as we want to hook a root node to the low parent. We need to connect root nodes to the new parent to avoid unhooking parents as happened in Figure 4.3. The loop then continues from step 4 with the new parent values.

The atomic operation in line 8 does more than just comparing two values and then swap. It finds the parent value of the first argument. It then compares this with the second argument, and if they are equal it sets the third argument as the new parent value. In Algorithm 5 the first and second argument are the same value high. The comparison is therefore between the parent of high and high. This gives the atomic operation a second function, where the high value needs to be a root node for the atomic operation to be successful. If the atomic operation succeeds then low is set as the new parent for high. If high is not a root node the atomic operation wont write and Algorithm 5 continues.

In Figure 4.2 we saw how Hook-non-atomic failed due to a race condition when two threads were trying to change the same parent value of node 2. Figure 4.4 shows how Hook-atomic handles the situation. Thread A gets the edge (2, 0), and thread B get the edge (2, 1), both marked in green in the left graph. Every node is initially its own parent. Now both thread A and B tries to change the parent value of node 2 since both node 0 and 1 have lower integer value then node 2. Thread A writes before thread B, and node 0 is set as the parent for node 2. Since the parent node of node 2 has changed B's atomic write operation will fail, and the parent values are updated. Thread B will now try to connect node 0 and node 1. This is represented in the middle graph, where the green edge now is between node 0 to node 1. As node 0 has a lower integer value node, 0 will be set as the parent of node 1. The result is one component with three nodes, shown to the right. This is the desired result Hook-non-atomic failed to deliver.

The second problem for Hook-non-atomic was that it might disconnect compo-

Figure 4.5: Hook-atomic avoid separating a component, where Hook-non-atomic would.

nent as shown in Figure 4.3. To handle this problem Hook-atomic ensurers that we only connect a root node to another node. When a component has a higher tree height than two Hook-atomic traverses up the parent structure to get to a root node. In Figure 4.5 we show how Hook-atomic handles the same example that Hook-non-atomic separated. Hook-atomic will move up the parent structure, until node 0 and node 1 are the nodes being compared. At this point node 1 is set as a child of node 0.

It follows that unlike Hook-non-atomic the result of Hook-atomic algorithm is always correct. But a downside is that we have a loop that can run several times as long as the atomic write is unsuccessful. This may happen in situations where several edges point to the same node, and multiple threads tries to change the parent of this node. Higher tree structures can also cause the atomic write to fail. This can happen when components containing more than one node are connected together.

4.2.3 Compress

The compress algorithm is an important tool in Groute's Connected components, Algorithm 3. In some versions of union-find the compression is done when connecting two components, one example is the Rem's algorithm in Section 2.4. To get the best performance from the Hook algorithms it is optimal to keep the parent structure at a tree height of two. When a tree height of one or two Hook-non-atomic cannot disconnect components and Hook-atomic is faster as the atomic write operations don't fail as often. When any of the Hook algorithms are complete the parent structures might have a hight larger than two. That is why the compress algorithm run after each use of the Hook algorithms. The steps for the

compress algorithm are shown in Algorithm 6.

The goal of the compress algorithm is to ensure that every node has a hight of at most two. The left graph in Figure 4.6 is a possible outcome after a Hook algorithm. Here node 3 has height three and it follow that the parent of node 3 is not its own parent. To solve this the compress algorithm shortens the height of this tree and moves node 3 up one level. It does this by changing a nodes parent to its parents' parent. In Figure 4.6 this is marked with the green edge from node 3 to node 0. Node 0 is the parent of node 2, so node 0 becomes the parent of node 3. In the right graph node 0 is the parent of every node.

Algorithm 6 Compress(parents)

```
1:  tid = parents.get_wid()
2:  if parents.read(tid) == tid then
3:      return
4:  end if
5:  p = parents.read(tid)
6:  pp = parents.read(p)
7:  while p ≠ pp do
8:      parents.write(tid, pp)
9:      p = pp
10:     pp = parents.read(p)
11: end while
```

The input to the Compress algorithm is not the same as for the Hook algorithms. While the Hook algorithms are executed over all edges, Compress is ran over all nodes in the graph. The distribution of the nodes is done by assigning as many threads as there are nodes in the graph. Then the thread ID is the same as the node ID. This is the first line in the algorithm. In line 2 the algorithm check to see if this node is a root node. If this is the case the thread exits the algorithm, there is no need to compress this type of node. If the node is not a root as the thread continues executing and retrieves the value of its parent and its parents' parent i.e. grandparent, in lines 5 and 6. In lines 7 to 11 a while loop that performs the compression. The loop continues as long as the parent and the grandparent are different. In the loop the grandparent is set to be the parent of the node, this is the compression. Then the parent of the grand parent is set as the new grandparent. This is then repeated until the new parent and the new grandparent have the same value.

Figure 4.6 shows one run through the compress algorithm, focusing on node 3.

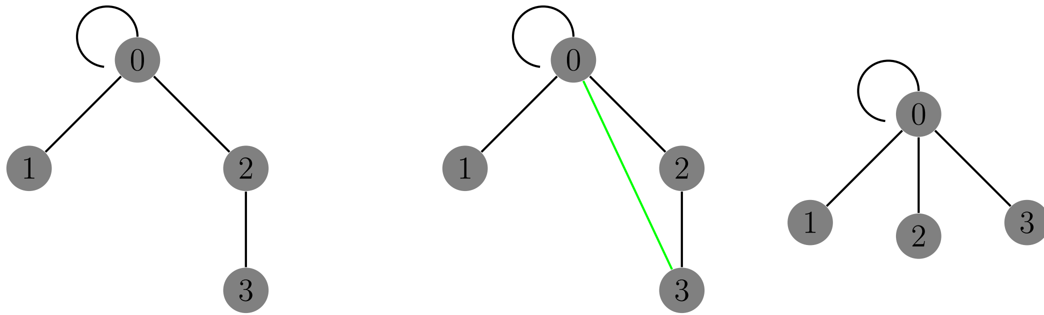


Figure 4.6: Compress algorithm compressing a parent tree with a height of 3

Node 3 is the node set in line 1. Node 2 is the parent of node 3, so node 3 isn't its own parent. Node 2 is then set as parent and node 0 as grandparent, lines 5 and 6. When the loop starts it is first determined that node 2 and node 0 are not equal, hence the loop runs at least once. Inside the loop node 0 is set as the parent of node 3, and the parent value is updated to 3. The parent of node 0 is node 0, so when the loop looks at the parent and grandparent that are now equal. The end result is the graph to the right in Figure 4.6.

The compress algorithm following Hook-non-atomic does not necessarily compress the whole parent structure, as it does after Hook-atomic. When performing this compression an upper-bound is given for which nodes to perform compress on. Only nodes from number zero to the upper-bound on node ID are compressed. This upper-bound is the highest valued node from the input segment and thus decreases the number of nodes that handled. This follows as there are no edges connecting nodes above the upper-bound to the nodes below.

The compress algorithm has two jobs. It improves the work conditions for Hook-non-atomic such that more of the work done by Hook-non-atomic is constructive and not destructive. The second part that it improves is the runtime of Hook-atomic by compressing the tree structure such that the atomic write doesn't fail as often.

4.2.4 Merge

Now that each device has a parent structure created from its assigned edges do we merge these together into the final solution. The second while loop in Algorithm 3 is where this merging happens. The merge algorithm uses Hook-atomic, Section 4.2.2, to connect components from these parent structures. The input is now edges from a parent structure created on another device. Merge only runs if more than

one device is used.

If device 1 is merging its solution with the one from device 2, then device 1 run Hook-atomic operations that take edges from device 2's parent structure as input. Device 1 is connecting components, using edges from Device 2s' parent structure and add this to its own parent structure. Thus components that were not connected as necessary edges was distributed to different graphs can now be connected. On device 2 node 0 is the parent of node 1, this is sent to device 1 as edge(0, 1). There are sent as many edges as there are nodes. Not all devices merge other devices parent structure, they just send their parent structure and are finished. The merging continues as long as there are devices receiving edges from other devices.

4.2.5 Final-compress

The last step for a device in Algorithm 3 is to prepare its local result for communication, as the result from one device is sent to another for collection into the final solution. The Final-compress algorithm shown in Algorithm 4.2.5, checks the parent structure on a device. If it is compressed, then the device is done. If the parent structure is not compressed, then the structure is compressed. The compression is performed by the Compress algorithm, presented in Section 4.2.3. This step is necessary in Groute as a solution with a higher parent structure than two is not recognized.

Algorithm 7 Final-compress(parents

1: **if** is.compressed() **then**
2: return
3: **end if**
4: Compress

4.3 Implementation of Rem in Groute

Rem's algorithm for CONNECTED COMPONENTS was presented in Section 2.4. We now describe how this algorithm has been implemented in Groute. Our goal when doing so is to get experience with Groute and see how hard it is to use it together with new algorithms. We are also interested in comparing the performance of Rem's algorithm with the existing algorithm in Groute. The result of these comparisons is presented in Chapter 6.

Algorithm 8 Rem in Groute

```
1: while Pending input_segement do
2:     Rem
3: end while
4: while Pending merge_segment do
5:     Merge-Rem
6: end while
7: Final-compress
```

To implement Rem's algorithm in Groute we use the code structure from Algorithm 3. Algorithm 8 shows the resulting algorithm in Groute. For communication purposes we need the two while loopsthat were also used in Groute's CONNECTED COMPONENTS program for communication purposes. The first while loop creates the parent structure on each devices from edges received from the input segment. The second while loop collects solutions from all devices into the final solution. The main change is that the for loop that iterates over Hook-non-atomic and compress has been removed. This can be done as Rem's algorithm compresses while building the tree structure and also uses atomic operations for merging. We keep the Final-compress algorithm that also compress since this is required to reuse this method. However, the merge algorithm in the second while loop use Rem's algorithm when building the final solution.

4.3.1 Adjusting Rem's algorithm for implementation

In Algorithm 9 we show Rem's algorithm as it is implemented in Groute. Since Hook-atomic and Rem have similar structure and input there are not many changes from from the version presented in Section 2.4. In line 1 how edges are retrived is copied from the Hook operations. The important change is that in the atomicCAS operation in line 15, the first argument is now ry, the node with the highest valued parent. In the Rem code presented in Section 2.4 there is used a read parent operation in this argument, as this is included in Groutes atomicCAS the node ry is used.

4.4 Experience with Groute

Our experience with Groute is impacted by the fact that there were no documentation available on how to apply Groute and implement with it. Where the

Algorithm 9 Rem(parents, edges)

```
1:  Edge e = edges.getEdge(threadID)
2:  rx = e.u
3:  ry = e.v
4:  prx = parents.read(rx)
5:  pry = parents.read(ry)
6:  while prx ≠ pry do
7:      if prx > pry then
8:          tmp = prx
9:          prx = pry
10:         pry = tmp
11:         tmp = rx
12:         rx = ry
13:         ry = tmp
14:     end if
15:     result = parents.write_atomicCAS(ry, pry, prx)
16:     if result ≠ pry then
17:         pry = result
18:         continue
19:     end if
20:     ry = pry;
21:     pry = parents.read(ry);
22: end while
```

CONNECTED COMPONENTS code in Gunrock alluded us was the code surrounding and including Groutes CONNECTED COMPONENTS implementation comprehensible. Swapping code to implement Rem was a straight forward procedure, as both Hook and Rem use integer values to ID nodes and select which node to set as parent. When we stepped up to the communication layer of the code was the lack of documentation more noticeable. We found enough understanding of the communication to get an idea on how the CONNECTED COMPONENTS implementation would communicate. Looking at the possibility of implementing a new algorithm in the Groute system was a more complicated step that we did not take.

The overall system Groute provided seem to be the necessary function that would allow us to create fast functional multiple GPU programs without the need to worry about optimizing communication and memory. Though the lack of documentation does make it harder to take advantage of Groute.

Chapter 5

CUDA Implementation

In Chapter 4 we describe a CONNECTED COMPONENTS implementation in a runtime environment. The runtime environment is used to run CONNECTED COMPONENTS with multiple devices and handles communication, memory and synchronization. In this chapter we describe an alternative CONNECTED COMPONENTS implementation that do not use a runtime environment, but is implemented in CUDA. Our goal is to get a comparable experience between working in Groute and CUDA. We show a way to handle multiple devices, manage communication between host and devices, and between devices, handle memory for host and all devices. We implement both Rem and Hook-atomic algorithms, and the opportunity for an iteration of Hook-non-atomic.

5.1 Application structure

Algorithm 10 show the CONNECTED COMPONENTS implementation made in CUDA. The algorithm starts by distributing parent lists and edge list to the devices in lines 1 to 4 from the input graph. The while loop in lines 5 to 16 connects components with the edges from the edge list. This loops as long as there are edges to execute on. The second while loop merge parent trees from devices into one final solution. In line 21 all parent trees have been merged into one complete solution, which is then sent to the host. Lines 8 and 9 is a Hook-non-atomic iteration and can be used or excluded depending on what type of version is used. In lines 10 and 18 use either Rem or Hook-atomic as the connective algorithm. Compress in lines 11 and 19 is only used if Hook-atomic is used.

The for loop in lines 1 to 4 in Algorithm 10 distributes the parent list and edge list from the host to each device by looping through all devices initiating the data

Algorithm 10 Connected components in CUDA

```
1: for Each device do
2:     Send parent list
3:     Send edge list
4: end for
5: while is.work do
6:     for Each device do
7:         if Use Hook-non-atomic then
8:             Hook-non-atomic
9:             Compress
10:        end if
11:        Rem or Hook-atomic
12:        Compress
13:    end for
14: end while
15: while is.merge do
16:     Rem-merge or Hook-atomic-merge
17:     Compress
18: end while
19: Solution retrieved from device to host
```

transfer with cudaMemcpyAsync. The data is transfer from the hosts memory to each device memory, storing it in the device global memory. The distribution is done asynchronous as the host continues to the next step after starting the data transfer to a device. All devices receive the same parent list, where each node is a root node. The edges are divide equally between all devices. All data transfers from the input graph are done in this step.

In the while loop lines 5 to 14 the connecting component operations are called. Only one of the four versions shown in Chapter 4 are called at a time. The while loop runs as long as there is work on at least one device. Work is true if a kernel call to a device occurred in the previous iteration, all devices have to be done before the host can continue from this loop. Hook-non-atomic and compress in lines 8 and 9 can be excluded from running when the algorithm executes only atomic versions. The compress operation in line 12 is only used with Hook-atomic, not for Rem as the Rem is compressing and it is not needed. Distribution of threads for kernel calls are calculated from the number of nodes in the graph. If there are more available edges than nodes on a device, then the number of edges is set to the number of nodes in the graph. If there are less edges than nodes left on

the device are the number of threads set to the number of edges left to execute on. Using the number of nodes as a basis for number of threads is chosen for two reasons. There is a limit to how many threads that can be used in this application. As the number of edges in some of the graphs used in test in this thesis is higher than the maximum number of threads the application can use we divide the edge list by using fewer threads but more kernel calls. In this while loop the edge list is separated into smaller chunks by using the number of nodes as the number of threads. After each iteration an offset is increased such that together with thread ID all edges have been added by the device. For both Hook versions there are compress operations after each Hook operation, to allow this compress to be called between Hook operations is it necessary to divide the problem size such that we get more iterations of this while loop and get this Hook-compress situation.

The while loop in lines 15 to 18 loops a set number of steps based on the number of devices in use. The loop is skipped if there is only one device in use. In this while loop devices either send their parent list to another device, receive a parent list from another device or wait for next iteration. A device waits if there are no devices to either send or receive the parent list. This communication step happens before either Rem-merge or Hook-atomic-merge merge a received parent list with the local parent list.

5.1.1 Changes for implementation

There are made changes to algorithms mentioned in Chapter 4, the changes are only minor variations directly influenced by how we chose to implement in CUDA. As the changes are syntactic they are similar for algorithms that do the same task. For all algorithms there are a change to how parent and edge values are accessed. As they are stored in integer pointers on each device, we use a call to the pointer to access the values, parents[index] is the parent value to the node with index value.

Connecting algorithms

Algorithm 11 show the new version for Hook-atomic. The input has changed from an edge list to two list holding the node value for an edge and an offset variable. An edge is found by the index in the edges list, edgesV[index] and edgesU[index]. The offset variable is used to ensure that all edges are used, when a kernel call is made only part of the edge lists are used. The number of threads currently called are stored in the offset variable so that in the next kernel call threads can add this offset value to their ID such that new edges are used. In line 1 in Algorithm 11 the offset is added to the threadID to obtain the correct id to access a new edge.

Algorithm 11 Hook-atomic(parents, edgesV, edgesU, offset)

```
1: tid = threadID + offset
2: if [edgesU[tid] ≠ edgesV[tid] then
3:     pu = parents[edgesV[tid]]
4:     pv = parents[edgesU[tid]]
5:     while pu ≠ pv do
6:         high = pu > pv ? pu : pv
7:         low = pu + pv - high;
8:         prev = atomicCAS(parents[high], high, low)
9:         if prev == high || prev == low then
10:            break;
11:        end if
12:        pu = parents[prev]
13:        pv = parents[low]
14:    end while
15: end if
```

For Hook-atomic and Rem the call to atomicCAS is different from how they were in Groute, now we use that standard atomiCAS operation, line 8. For Hook-atomic to reach a root in this atomic operation we look at the parent of high in the call, such that high must be equal to parent of high for a successful write. For Rem this atomic operation is the same as in the original version, see Section 2.4.

Merge algorithms

Algorithm 12 show the first lines in the Rem-merge algorithm. The input also changes for the both Hook and Rem merge algorithms, the change is the same for both. When two devices merge their parent list one device sends their parent list to the other device. The parent list is stored such that the index of the parent list is the ID of a node, and the value at this index is the parent of that node, this gives us the edge between these two nodes. This is used in lines 1 and 2, parentsEdges is the parent list from the other device. After the nodes are collected from parentsEdges and set to rx and ry, then the algorithm continues as do in Rem and Hook-atomic.

Algorithm 12 Rem-merge(parents, parentsEdges)

1: rx = parentsEdges[threadID]
2: ry = threadID
3: prx = parents[rx]
4: pry = parents[ry]
5: ...

5.2 Communication - data transfer

In this section we look at the two communication steps in the CUDA implementation. The first step is distributing needed data to the devices, this includes a complete parent list of all nodes for each device and list of edges which are distributed among the devices. In the second step have each device calculated a solution based on the edges they received, these solutions must be collected and connected together and sent back to the host.

5.2.1 Distributing graph

The first communication step is distributing the graph to all devices. Every device gets a parent list where all nodes are root nodes. All edges are divided between all devices, an edge exist only on one device. The content that is distributed is the same for this implementation as it is in the Groute implementation, the difference is how the edges are distributed. In this implementation all edges are distributed at once to all devices. All devices have all their dedicated edges allocated to their memory before they start connecting components. In Groute the edges are divided into smaller segments that are distributed a segment at a time to devices, a device connects components on their current input segment, when all edges in the segment are handle they receive a new segment with new edges.

5.2.2 Merging solutions

The second communication step in this application is to gather partial solutions from all devices and merge them together as one final solution. Devices on the Google Cloud server are structured in two groups where devices in each groups can send data to each other, but must go through host two send data to the other group, this setup is described in Section 2.5. Device to device communication is faster than device to host communication thus we want to use these groups for optimized communication. Figure 5.1 show the merge-communication structure used for the

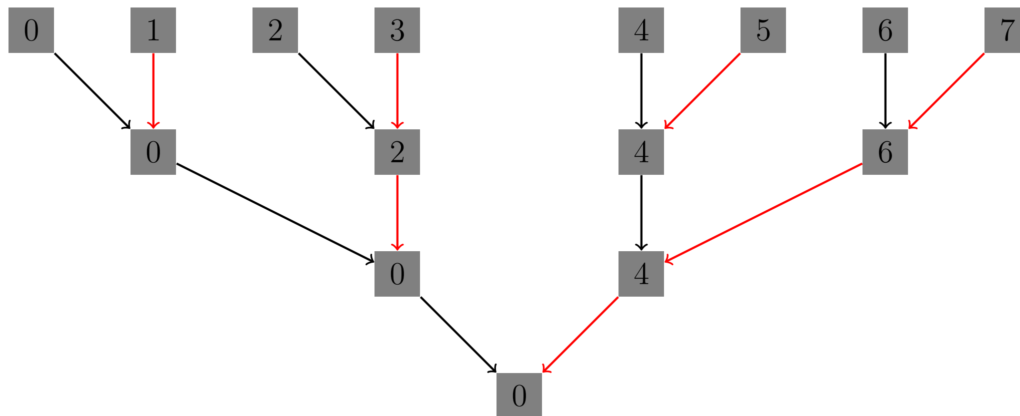


Figure 5.1: Merge communication between devices as parent lists are merged into one on device 0, red arrows show devices that send their parent list.

second communication step with eight devices. Device 0-3 is one group of devices with connections between them, device 4-7 is the other group. The communication between devices are kept within the groups for as long as possible, before the host is used for the last step.

When two devices communicate one device send its parent list to the other device. Devices that sends their parent list are done and don't have any more tasks to do. Devices that receives a parent list merge this list with its own parent list before it continues with further communication. In Figure 5.1 we show the steps in communications, red lines show devices that send their parent list to another device. We show that devices connected in groups communicate with each other before the last step where device 4 send its parent list to device 0, this is done through the host.

5.3 Handling multiple devices

In this section we describe steps taken to handle more than one device when programming in CUDA. CUDA has integrated functions to handle multiple devices, two we focus on are setCudaDevice and streams, described in Section 2.2. Both functions are used to determine which device that is currently active for the host to send instructions or operations to, such as copying data or kernel calls. Therefore we loop through each device when preparing calls to devices, in lines 1 to 4 and 6 to 13 in Algorithm 10. This is also used in the while loop in lines 15 to 18, in this loop every device is not used each iteration as devices that sends transfer their

parent list are done.

Memory management

Techniques used when programming on one device are transferable when we move to more devices. But where one would create a pointer to refer to some data are we creating lists of pointers to represent data. To allow scaling the number of devices we create a list where the number of pointers corresponds to the number of devices. Thus, each device have a pointer for their data set that corresponds to their device ID. In this implementation we encountered the need for variables, as the values of these variable varies from device to device we stored these in lists to. The advantage of this technique is that it makes handle memory easier but might not be very effective for speed or memory usage.

5.4 Experience from multi-GPU programming

In this section we discuss our experience gained from working on a multiple GPUs program in CUDA. There are a few subjects we need to consider when we create a CUDA program for multiple devices, how to handle devices, synchronization between devices and how to share memory. When we think about these subjects we encounter a point where a level of optimization is set. Creating a program where optimization is not important make this task easier, as would be expected. We can now use a loop to manage these subjects, by going through each device and perform a task. When optimization is important there seems to be a need for an overhead that demands more intricate solutions. The reason for this is that there is much to gain by controlling and using the full memory bandwidth such that operations on host or device must wait as little as possible. At this point a library or runtime environment that have solved this is advantageous.

Creating a program in CUDA that works on multiple devices was surprisingly intuitive with a CPU parallel programming background. This is surprising as there are multiple levels of parallelism when we move to multiple devices. Though moving from a program that works to a fast program that has highly optimized communication and memory handling is a more complicated step.

Chapter 6

Experiments

In this chapter we experiment on the performance for Groute and the CUDA implementation, comparing their run time. We look at the efficiency in scaling for both and at how much speedup they achieve. We describe the graphs used for the experiments, the computers used for these experiments are described in Section 2.5. We look at the performance of the algorithm already implemented in Groute and compare it with the Rem's algorithm implementation to show how Rem's algorithm perform in parallel.

6.1 Graphs

In this section we take look at the graphs used for experiments. We have selected a variation of graphs to allow a better range for the results finding general performance indications. Graphs varies in size, average degree and how they are generated, they are described in Table 6.1. These graphs are collected from the 10th DIMACS Implementation Challenge [7].

2D Dynamic simulation graphs

2D Dynamic Simulations graphs are meshes taken from individual frames of a dynamic sequence which resembles adaptive 2D numerical simulations [7]. These graphs has a low average degree, all three has an average degree of 3.

Citation network

These graphs represent real-world social networks [7]. They have a higher average degree, 73.9 for coPapersCiteseer and 56.4 for coPapersDBLP.

Graph	Vertices	Edges	Category	Components
hugebubbles-00000 [12]	18 318 143	27 470 081	2D Dynamic sims	1
hugebubbles-00010 [12]	19 458 087	29 179 764	2D Dynamic sims	1
hugebubbles-00020 [12]	21 198 119	31 790 179	2D Dynamic sims	1
coPapersCiteseer [10]	434 102	16 036 720	Citation network	1
coPapersDBLP [10]	540 486	15 245 729	Citation network	1
delaunay_n23 [7]	8 388 608	25 165 784	Delaunay	1
delaunay_n24 [7]	16 777 216	50 331 601	Delaunay	1
rgg_n_2_22_s0 [11]	4 194 304	30 359 198	Random Geometric	5
rgg_n_2_23_s0 [11]	8 388 608	63 501 393	Random Geometric	5
rgg_n_2_24_s0 [11]	16 777 216	132 557 200	Random Geometric	2
af_shell10 [7]	1 508 065	25 375 910	Sparse matrix	1
audikw_1 [7]	943 695	38 354 076	Sparse matrix	1
nlpkkt120 [7]	3 542 400	46 651 696	Sparse matrix	1
nlpkkt160 [7]	8 345 600	110 586 256	Sparse matrix	1
nlpkkt200 [7]	16 240 000	215 992 816	Sparse matrix	1
nlpkkt240 [7]	27 993 600	373 239 376	Sparse matrix	1
asia.osm [7]	11 950 757	12 369 181	Street network	1
europe_osm [7]	50 912 018	54 054 660	Street network	1
germany.osm [7]	11 548 845	12 369 181	Street network	1
kron_g500-logn20 [7]	1 048 576	44 619 402	Synthetic	253 380

Table 6.1: Graphs used for testing, with the number of vertices, edges and components

Delaunay Graphs

Delaunay graphs are generated as Delaunay triangulations of random points in the plane [7]. Both Delaunay graphs have an average degree of 6.

Random Geometric graphs

Random Geometric Graphs are graphs that vertices are random points in the unit square and edges connect vertices with a fixed Euclidean distance. This distance ensure that the graphs are almost connected [7]. The average degree for these graphs is between 14 and 15.

Expanded graph	Vertices	Edges	expanded	Components
rgg_n_2_24_s0exp10	167 772 160	1 325 572 000	10 times	20
nlpkkt240exp4	111 974 400	1 492 957 504	4 times	4
rgg_n_2_22_s0exp2	8 388 608	60 718 396	2 times	10
rgg_n_2_22_s0exp3	12 582 912	91 077 594	3 times	15
rgg_n_2_22_s0exp4	16 777 216	121 436 792	4 times	20
rgg_n_2_22_s0exp5	20 971 520	151 795 990	5 times	25
rgg_n_2_22_s0exp6	25 165 824	182 155 188	6 times	30
rgg_n_2_22_s0exp7	29 360 128	212 514 386	7 times	35
rgg_n_2_22_s0exp8	33 554 432	242 873 584	8 times	40

Table 6.2: Nine expanded graph for further testing

Sparse matrix graphs

These sparse matrix graphs are taken from the Florida Sparse Matrix Collection and converted to 10th DIMACS Implementation Challenge's graph format [7]. The average degree varied between these graphs, all nlppkt graphs have an average degree about 26, while af_shell10 is 33.7 and audikw_1 the second highest for all selected graphs at 81.3.

Street network graphs

The Street network graphs are undirected and strongly connected components from The Open Street Map road networks [7]. These graphs are from real-world street networks, thus they have planar tendencies. They also have the lowest average degree among the selected graphs, all street network graphs have average degree 2.1.

Synthetic graph

The synthetic graphs are generated with the Kronecker generator and contains self-loops and multiple edges [7]. This synthetic graph has the highest average degree among the selected graphs, at a average degree of 85.1.

Expanded graphs

We have expanded nine graphs for testing, we expanded known graphs to keep some expected characteristics but also achieve a bigger workload. Seven of them

$T1/(Tn * n)$	$T1/Tn$	$T1/Tn$
Fixed problem size.	Scaled problem size.	Fixed problem size.
Figure 6.1: Strong scaling	Figure 6.2: Weak scaling	Figure 6.3: Speedup

are expanded to measure weak scaling for one to eight devices, increasing the work load at the same rate as the number of devices. The two other graphs are expanded to increase the work size for applications in this thesis, allow us to test on more than one billion edges. The expansion is done by taking a graph and replicate it a given number of times, where each replicate is a stand-alone version of the original graph where there are no connections between the replicates. Information about the graphs are showed in Table 6.2.

6.2 Test measurements

In this chapter we do experiments on our programs and we look to four tests to help us determine how a program is performing in these experiments. We want to determine how fast the program is and how it performs as we increase the number of devices. Run time refers to the time a program spends, this shows us the speed of a program on our test computers and is measured in milliseconds. There are two ways to determine how efficiently a program scale as the number of devices increase, strong and weak scaling. Strong scaling show how well the program scale as the number of devices increase and the problem size stays fixed. We find the efficiency for strong scaling by finding the time it takes the program to run on each given number of devices. We multiply the time the program spent with the number of devices used and divide the time it took the program to run on one device. In Figure 6.1 the equation for strong scaling, T is time and n the number of devices. For weak scaling we look at the scaling efficiency as the problem size increase as the number of devices increase. To find weak scaling efficiency we increase the problem size such that the work for each device stay the same when the number of devices is increased. Figure 6.2 show the equation for weak scaling, the time it takes for one device is divided on the time it takes on n devices, where the problem size is increased n times. Speedup shows how much faster a program is on n devices, the problem size is fixed. Figure 6.2 show the equation for speedup, if the time is halved when the program run on two devices compared to on there would be a speedup of two.

6.3 Groute experiemnts

In this section we look at the experiments on Groute and its CONNECTED COMPONENTS algorithm and with Rem. We show the run time comparison between the best version of Hook and the best version of Rem, with regards to the number of iterations with Hook-non-atomic. We describe the scaling efficiency Groute has as the number of devices increase. We also look at the speedup that Groute can achieve.

6.3.1 Hook-non-atomic

In Section 4.2.1 we look at the performance in Groute for both Hook and Rem with different number of iterations of Hook-non-atomic. Our goal is to find the number of iterations that yields the best performance for both versions. We use four graphs as a sample of our graph collection to show the tendencies with different number of iterations. We look the run time performance and the percentage each iteration performs compared to the best for each graph.

Hook-atomic with Hook-non-atomic iterations

Figure 6.4 shows the time each iteration take in the left plot, the right plot show performance by percentage. For graphs that have the best performance with one iteration we find that no iterations or more than one iteration is noticeable worse, performing below 80% compared with one iteration. For graphs that has the best performance with no iteration, the drop in performance with one iteration is not as significant, staying above 90%. For all graph the performance decrease when more than one iteration used. Groute's Hook implementation work generally best with one iteration of Hook-non-atomic.

Rem with Hook-non-atomic iterations

With Rem's algorithm we don't expect there to be better performance with Hook-non-atomic as there should be little advantage to use it. Figure 6.5 shows the results for Rem. Here the tendencies shift towards overall better performance with no iterations of Hook-non-atomic. The performance decrease as the number of iterations increase. This is expected as the pre-work Hook-non-atomic does for Hook-atomic do not decrease the work for Rem as much as it does for Hook-atomic.

Street network graphs is an exception for Rem, these graphs perform much better with one iteration of Hook-non-atomic. Figure 6.6 shows the results for

Figure 6.4: Running time and performance by percentage on Hook-non-atomic iterations using Hook-atomic in Groute

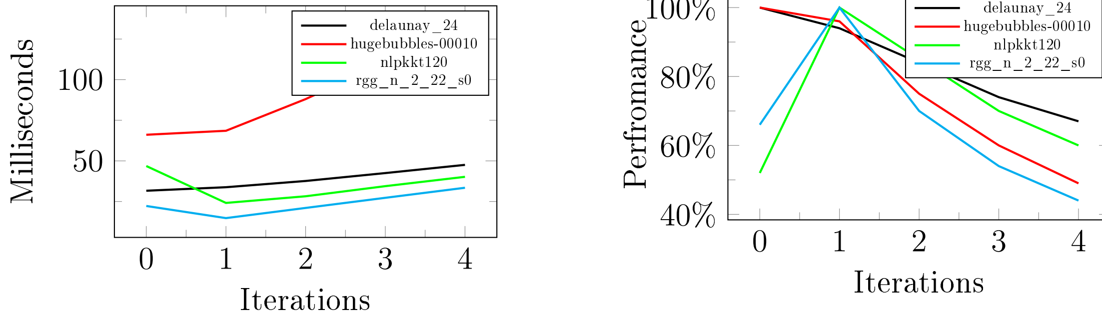
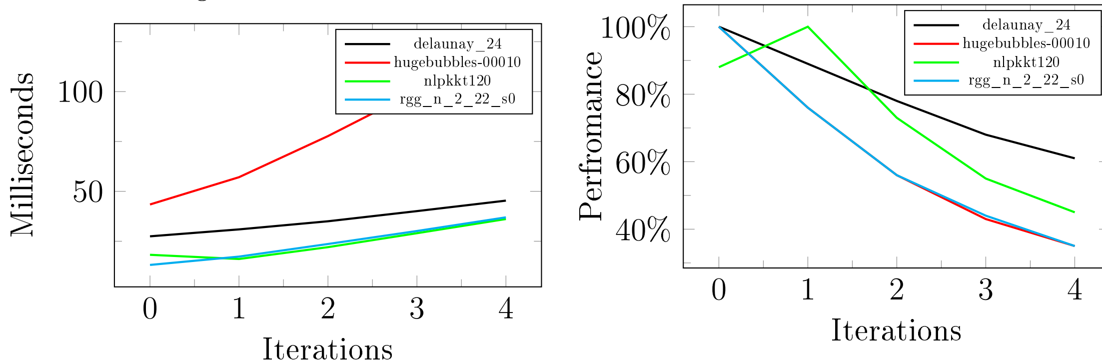


Figure 6.5: Running time and performance by percentage on Hook-non-atomic iterations using Rem

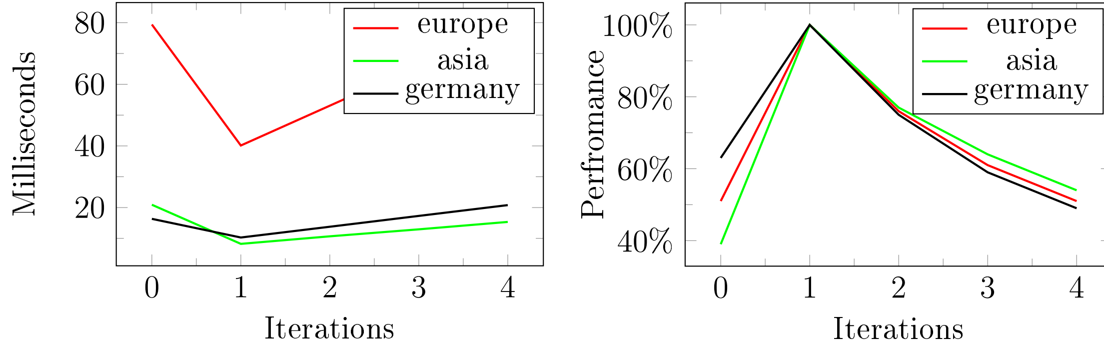


street network graphs, the left plot show the run time, the right show percentage of the performance. For these graphs the performance for no iteration of Hook-non-atomic is below 70% at best and below 40 % at the worst. Increasing to two iteration all graphs perform below 80% compared with one iteration. For street network graph the performance with Groute using Rem is better with one iteration of Hook-non-atomic.

Optimal number of iterations

From experiments we show that using either no or one iteration of Hook-non-atomic is the range where we achieve the best performance for both Hook and Rem versions in Groute. This range of iterations defines what we want to test in

Figure 6.6: Street network graphs performing better with one iteration of Hook-non-atomic



further testing when we increase the number of devices.

6.3.2 Rem and compress operations

Both Hook and Rem use a version of UNION-FIND for solving CONNECTED COMPONENTS. The Hook algorithms don't compress the parent structure when they connect nodes and components, Rem's algorithm do compress while connecting. Thus, compressing between handling segments don't decrease the run time for Rem as it does for the Hook.

Figure 6.9 show results from two graphs running Rem, with and without compress. On the rgg_n_2_24_s0 graph there is no difference between using compress and not using compress. Using compress or not is equally fast and scale the same. This is the same with and without an iteration of Hook-non-atomic. Running on nlpkkt160 compress slows down the run time. The difference is biggest on one and two GPUs. With an iteration of Hook-non-atomic it is 20% faster without compressing than when compressing. Not using compress is 13% faster with no iteration of Hook-atomic on one and two GPUs. From three GPUs the difference is much smaller, between 6% and 1% difference. In these tests we used compress in the Hook-non-atomic iteration, as not using compress increases the run time. Running on the nlpkkt160 graph, it is 30% slower to not use compress after Hook-non-atomic.

6.3.3 Run time perfromance Hook vs Rem

In this section we compare the run time performance between Hook and Rem in Groute. As we have seen there is a difference between no and one iteration of Hook-

Figure 6.7: rgg_n_2_24_s0 graph

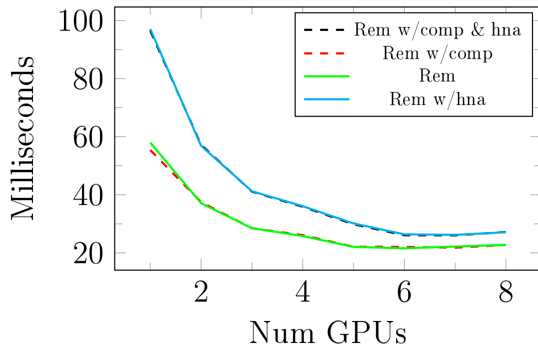


Figure 6.8: nlpkkt160 graph

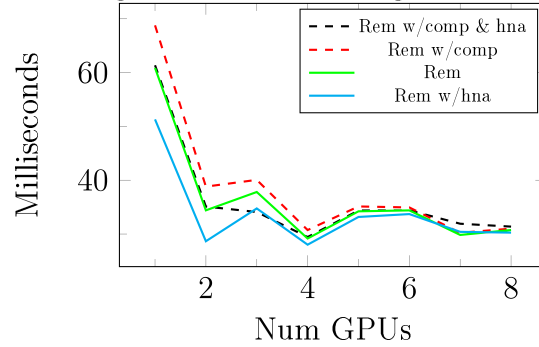


Figure 6.9: Comparison of Rem running with and without the compress algorithm in Groute

Figure 6.10: hugebubbles-00000

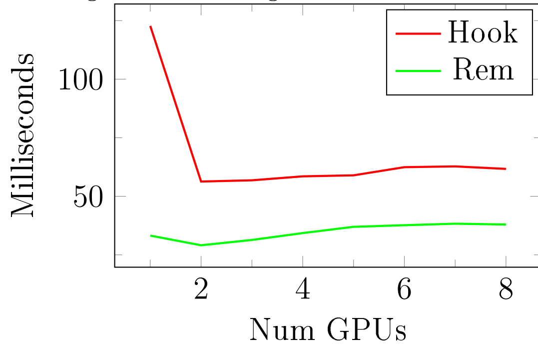


Figure 6.11: hugebubbles-00010

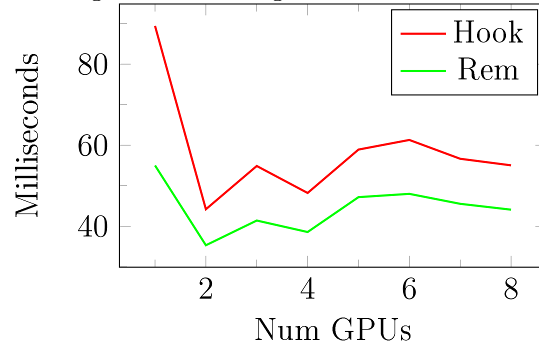


Figure 6.12: Run time on 2D Dynamic simulations graphs

non-atomic for both Hook and Rem, we use the number of iterations that gives the best performance for each graph when we compare Hook with Rem. Our goal is to see which algorithm perform best on Groute running on one to eight devices. We use Google Cloud server for these test, Google Cloud server is described in Section 2.5.

2D Dynamic simulation graphs

Figure 6.12 and 6.14 shows the running time for 2D Dynamic simulation graphs. On these graphs both Hook and Rem with no iteration of Hook-non-atomic are the best performing versions. The hugebubbles-00000 graph has the biggest difference

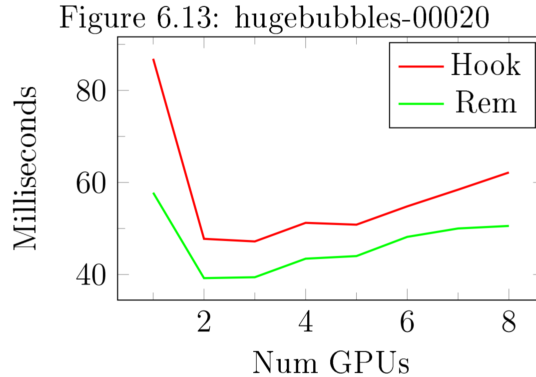
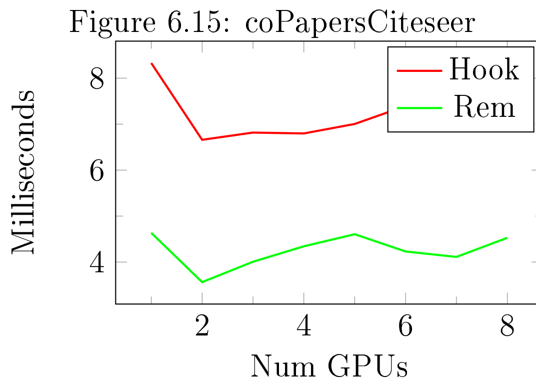


Figure 6.14: Run time on 2D Dynamic simulations graph



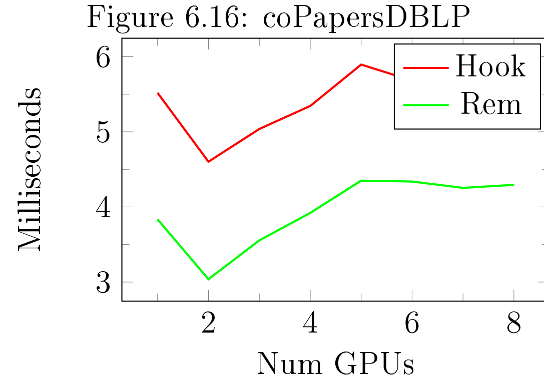
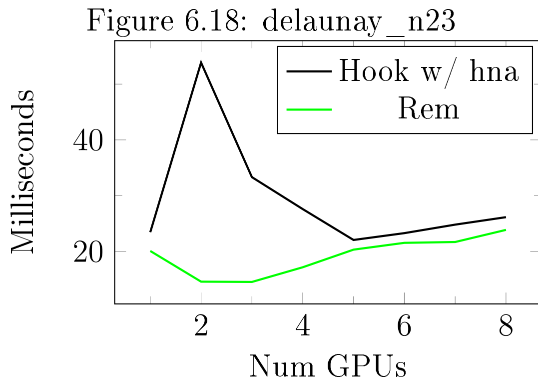


Figure 6.17: Run time on citation network graphs

in run time at one device, Hook is 269% slower than Rem on one device. At two devices the difference is at 97% slower, this goes down gradually to five devices where the difference levels out at around 60% slower. The other two graphs have s similar characteristics, but not as big of a difference. Hook is 63% slower on one GPU and levels out at 25-30% slower running hugebubbles-00010. And 50% slower on one device , levels out at 15% to 20% slower with hugebubbles-00020.

Citation network

Figure 6.17 show the results for citation network graphs, for tehse graphs are both versions better with no iteration of Hook-non-atomic. In both citation ntwork



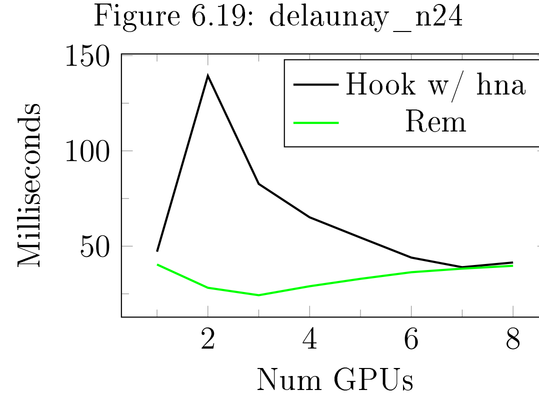


Figure 6.20: Run time on Delaunay graphs

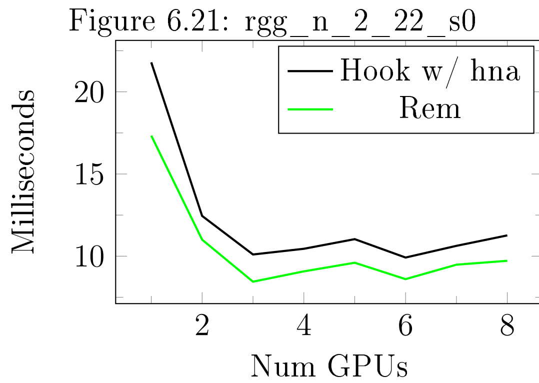
graph is Hook consistently slower than Rem. With the coPapersCiteseer graph is Hook between 31% and 51% slower variable difference when the number of devices is increased. For coPapersDBL is Hook between 87% to 52% slower.

Delaunay Graphs

For Delaunay graphs Hook with one iteration of Hook-non-atomic and Rem with no iteration the best versions. The results for these graphs are shown in Figure 6.20. For both graphs are Hook close to Rem in performance on one device, 17% slower for both graphs compared with Rem. Hook increase in run time when two devices are used, delaunay_n23 is 269% slower and delaunay_n24 is 395% slower. The time decrease from this peak as the number of devices are increased. With delaunay_n23 Hook is close to Rem at five devices, staying between 8% and 14%. For delaunay_n24 Hook is closer at seven and eight devices, only 2% and 4% slower. These peaks are even more extreme when Hook don't use an iteration of Hook-non-atomic.

Random Geometric graphs

Figure 6.23 and Figure 6.25 shows the results when running random geometric graphs on one to eight devices. The difference between Hook and Rem is bigger on one device before it levels out. Hook is 26% slower on rgg_n_2_22_s0, 46% slower on rgg_n_2_23_s0 and 57% slower on rgg_n_2_24_s0. The difference levels out as the number of devices increase, then Hook is around 15% slower on rgg_n_2_22_s0, around 23% slower on rgg_n_2_23_s0 and 16% to 30% slower



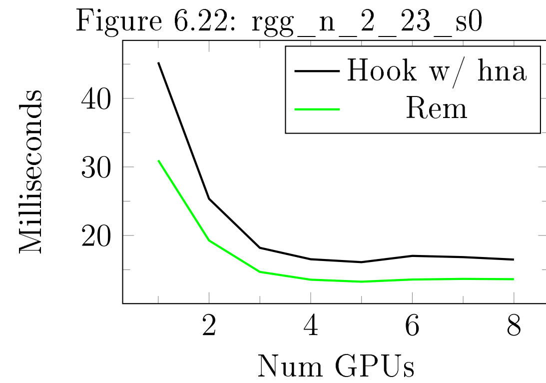


Figure 6.23: Run time on Random Geometric graphs

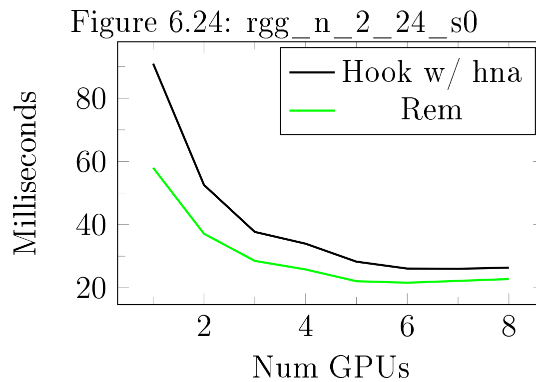


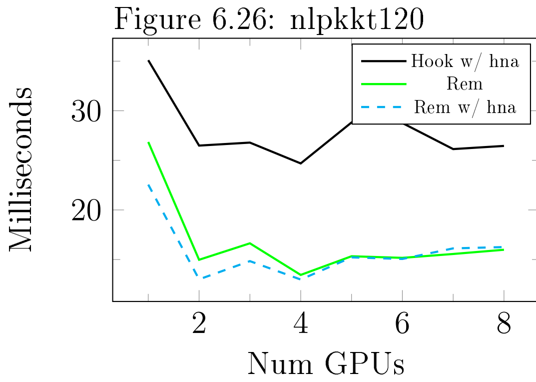
Figure 6.25: Run time on Random Geometric graph

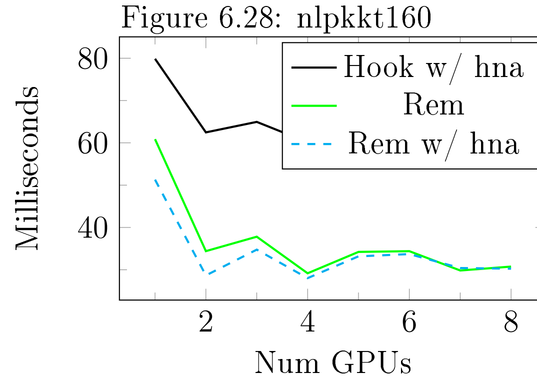
on rgg_n_2_23_s0.

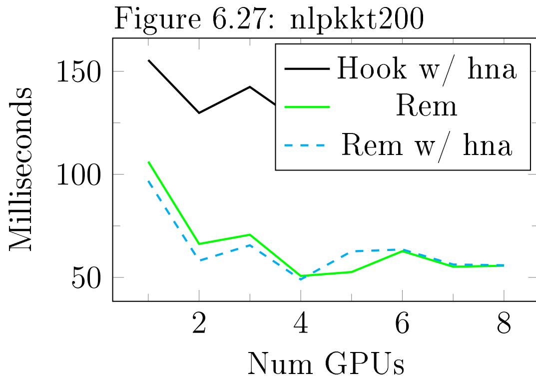
Sparse matrix graphs

Figure 6.30 and Figure 6.33 show the results for sparse matrix graphs. The spare matrices graphs are among the few graphs were Rem have a better performance with one iteration of Hook-non-atomic. The difference is not as significant as with the street network graphs. Hook is better with one iteration of Hook-non-atomic, except the af_shell10 graph, where no iteration is best. For nlpkkt graphs is Hook closer to Rem on one device, Hook is 50% to 60% slower. With more devices is Hook much slower than Rem, using more than twice the time running Hook.

The results for the smallest sparse matrix graphs are shown in Figure 6.33. For







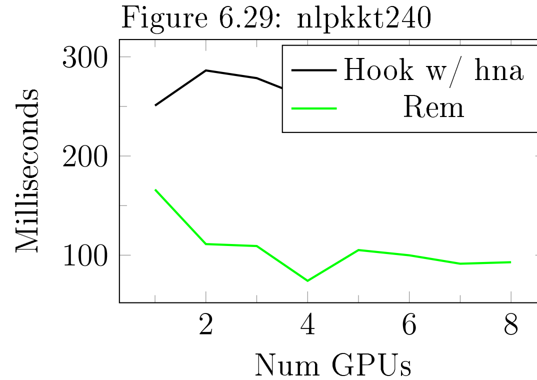


Figure 6.30: Run time for Sparse matrix nlpkkt graphs

af_shell is Hook 9% slower on one device. After two devices both Hook and Rem perform worse as the number of device increase, Hook is between 18% and 33% slower from two to six device, after this Rem's run time increase more, as Hook is only 15% slower. With audikw_1 Rem's run time is stable, decreasing from 13 ms to 10 ms from one to eight devices. Hook improves more from one to eight devices, Hook is 114% slower at one device and gradually goes down to 49‘5 slower as the number of devices increase to eight devices.

Street network graphs

In Section 6.3.1 we described the exception that Rem performed much better on street network graphs with one iteration of hook-non-atomic. We show the results for street network graphs in Figure 6.36 and Figure 6.37. The street network

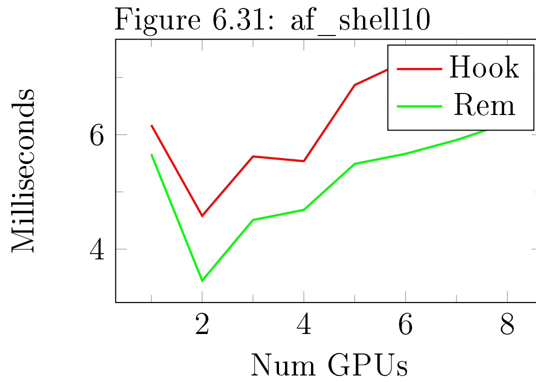


Figure 6.31: af_shell10

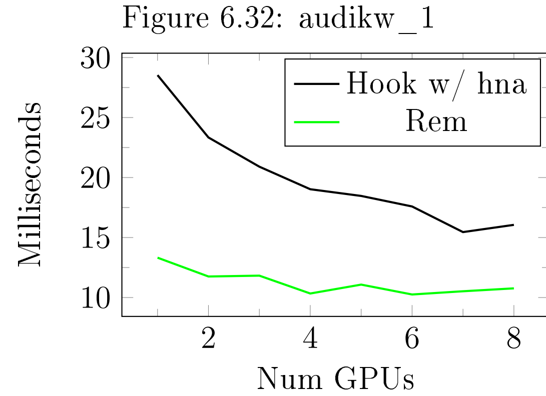


Figure 6.32: audikw_1

Figure 6.33: Run time on small sparse matrix graphs

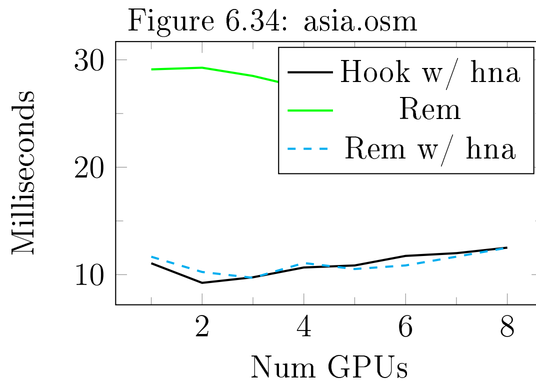


Figure 6.34: asia.osm

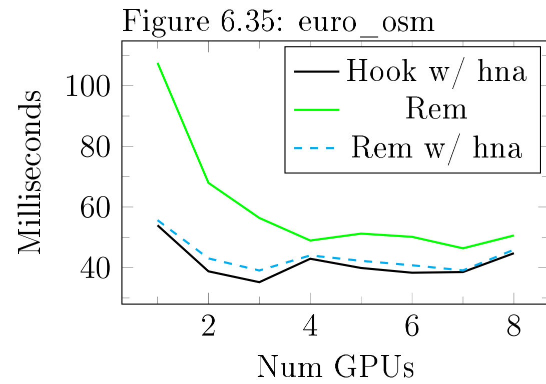


Figure 6.35: euro_osm

Figure 6.36: Run time on street network graphs

graphs are the only graphs where Rem is not faster than Hook on all every number of devices. There are variations whether Hook or Rem is faster, either is not more than 10% faster, varying between 0% and 6% difference.

Synthetic graph

Figure 6.38 show the run time result for the synthetic graph kron_g500-kogn20. Hook and Rem both perform best with no iteration of Hook-non-atomic. The difference between Rem and Hook is bigger at one device than other number of devices. Atone device is Hook 33% slower, after that it around 20% slower.

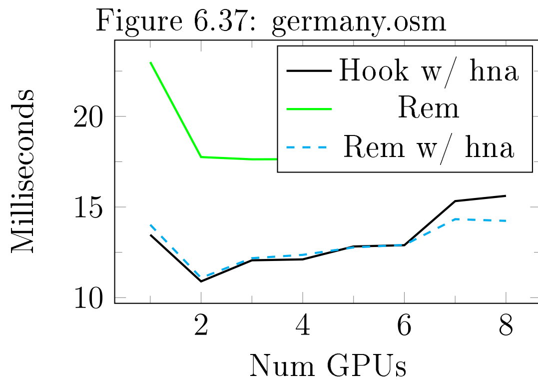


Figure 6.37: germany.osm

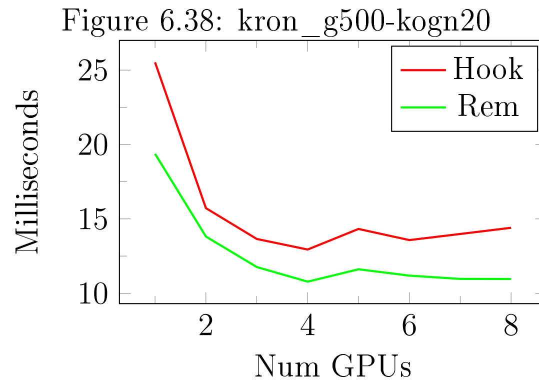
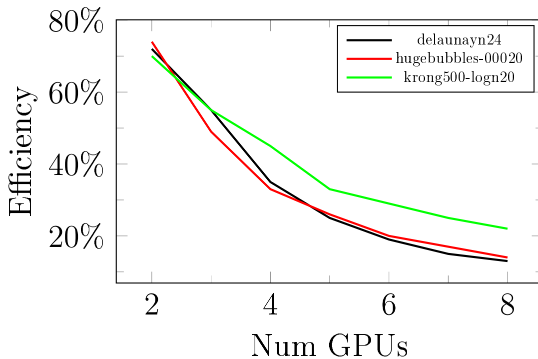


Figure 6.38: kron_g500-kogn20

Figure 6.39: Run time on street network and synthetic graphs



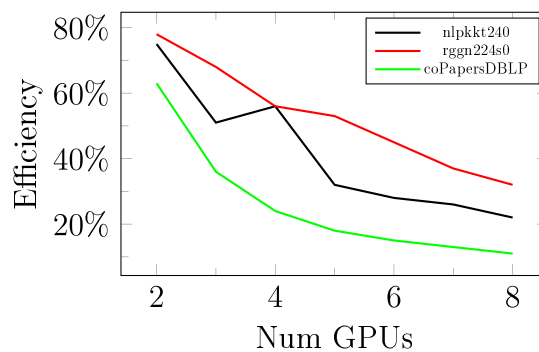


Figure 6.40: Strong scaling efficiency for Groute

6.3.4 Strong scaling

In this section we show the strong scaling efficiency for Groute running Rem on six graphs. Strong scaling shows the efficiency when the problem size stays the same and the number of GPUs is increased, at 100% efficiency the run time decrease at the same rate as the number of GPUs increase. 100% efficiency is unrealistic as we know there is overhead work that increases as the number of devices do. In the left plot we see that the efficiency running two GPUs is acceptable around 70%, but the efficiency drops significantly as the number of GPUs increase. In the right plot is the variation between the graphs more significant, two of the graphs perform as in the left graph, the rgg_n_2_24_s0 is decreasing slower, but the efficiency is still too low when the number of devices increase. The worst efficiency is on the

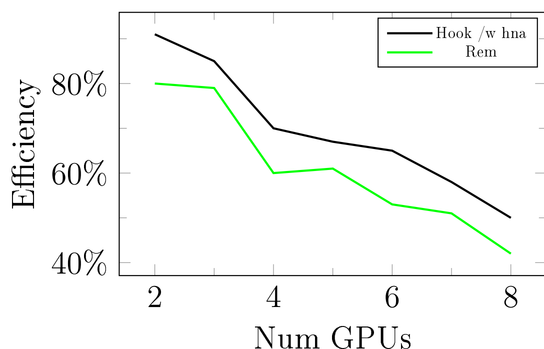


Figure 6.41: Weak scaling comparison

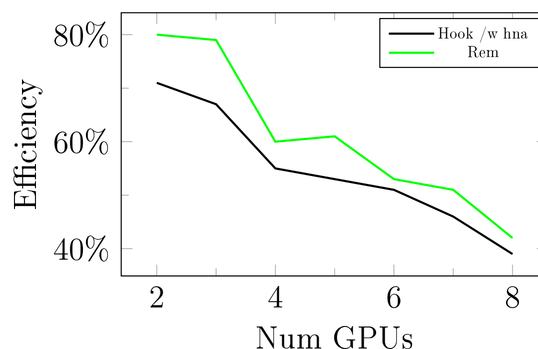


Figure 6.42: Weak scaling based on Rem

coPapersDBLP graph, this might be due to it being the smallest graph in these two plots, thus there is not enough work for the GPUs added to scale efficiently.

6.3.5 Weak scaling

In this section we look at the weak scaling efficiency of Groute. We want to see how efficiently Groute scales when we increase the problem size as we increase the number of devices. An efficiency of 100% is unrealistic as there is an increase in overhead work when the number of devices is increased. We expanded the rgg_n_2_22_s0 graph to have a scalable problem size that matches the number of devices used, expanded graphs are presented in Section 6.1. Rem with no iteration and Hook with an iteration of Hook-non-atomic are the best performing versions. Figure 6.41 shows how Hook and Rem scale individually where the efficiency is calculated based on the result on one device for each of the algorithms. We see that the original version in Groute scales better than with Rem, though we show that Rem is much faster. Figure 6.42 shows how well both scale with regards to Rem. On two and three devices is the efficiency good, not as good as the original was. From four devices the efficiency drops and is not as good as we would like.

6.3.6 Speed-up

We are in this section showing the speedup for Groute. With speedup we measure the increase in speed as the number of GPUs are increased, perfect speedup the rate of speed increase at the same rate as the number of GPUs are increased. We don't anticipate a perfect speedup as there is an overhead when we use multiple

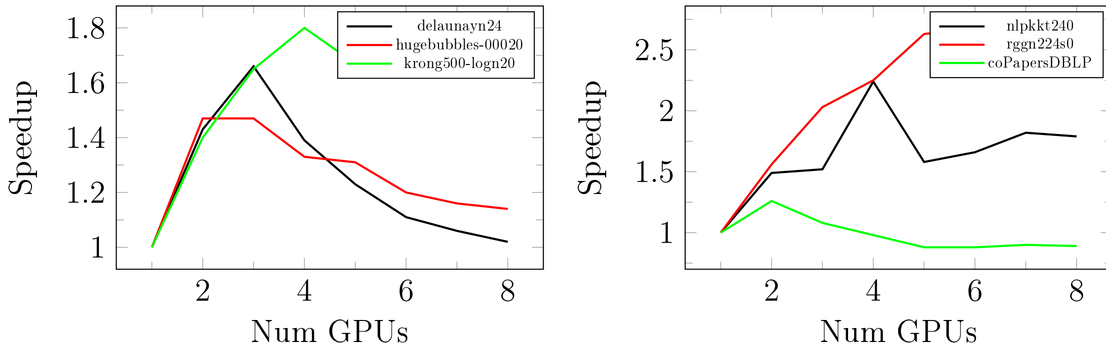


Figure 6.43: Speedup on Groute

Figure 6.44: nlppkt240exp4

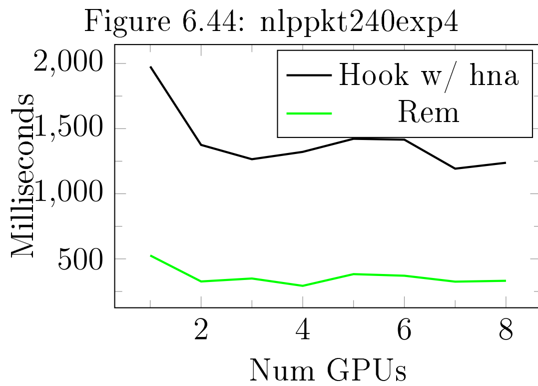


Figure 6.45: rgg_n_2_24_s0exp10

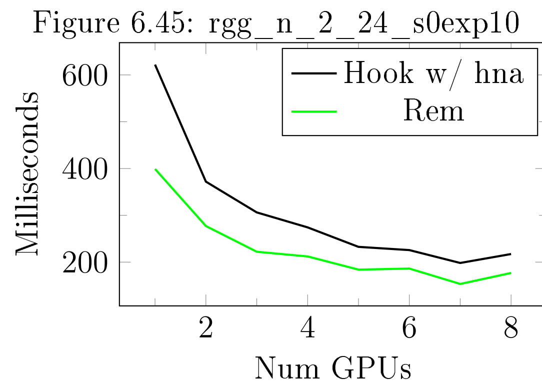


Figure 6.46: Run time for the expanded graphs

GPUs, such that there is more work when the number of devices increase. Figure 6.43 show the speedup for Groute running Rem with no iteration of Hook-non-atomic on six graphs. The speedup for Groute is not near perfect speedup, the best speedup achieved is with the rgg_n_2_24_s0 graph, at 2.6 on five GPUs. There is an increase in speed for all graphs running two GPUs, most graphs declines after three GPUs.

6.3.7 Expanded graphs

In this section we expand two graphs to increase the workload to a significant bigger size than previously tested graphs. The expanded graphs are described in Section 6.1. The goal is to see if a big increase in the workload gives increased

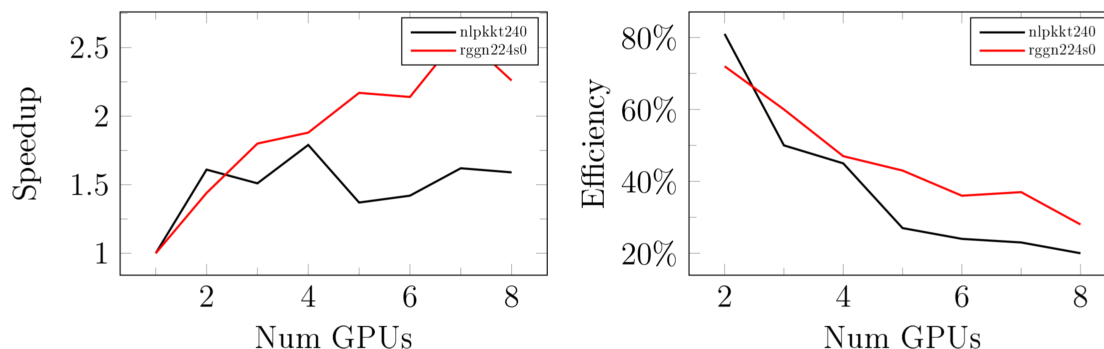


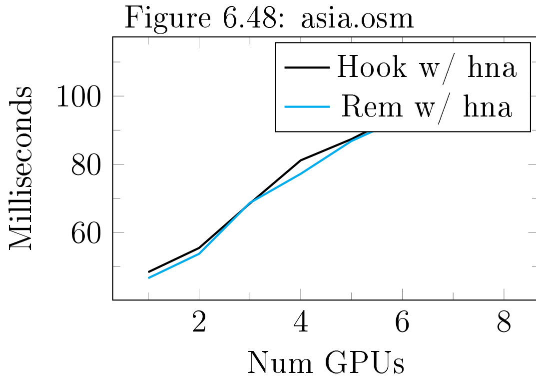
Figure 6.47: Speedup and strong scaling for expanded graphs

performance such as better scaling and speedup. A common trend for many of the graphs is that the improvement flattens out when the number of devices reach a point between three and six. We use Rem with no iteration of Hook-non-atomic when we test strong scaling and speedup on these expanded graphs.

Figure 6.46 shows the run time result for both expanded graphs. In the original nlpkkt240 graph Rem was 3.85 times faster than Hook running on one GPU, in the expanded graph Rem is 16.75 times faster. Rem is performing better when the graph size has been increased compared with Hook. Figure 6.47 show the speedup and strong scaling for the expanded graphs. For strong scaling we see the same tendencies as in the original graphs, the efficiency at two devices is between 70% and 80%, and declines as the number of GPUs increase. We do not get a significant increase in speedup when the graph size is increased significantly, but the peak for rgg_n_2_24_s0exp10 is higher at seven GPUs, the peak in the original is at five GPUs.

6.4 CUDA implementation experiments

In this section we describe the experiments done with the CUDA implementation in Chapter 5. We show how the run time compare for Hook and Rem, how the run time on this implementation compares with the run time on Groute. We show how this implementation scale and what speedup it achieves.



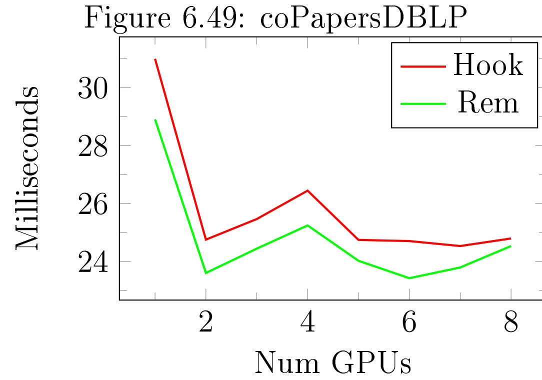
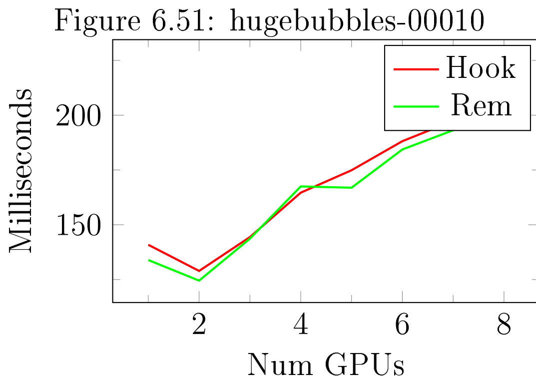


Figure 6.50: Run time performance on CUDA



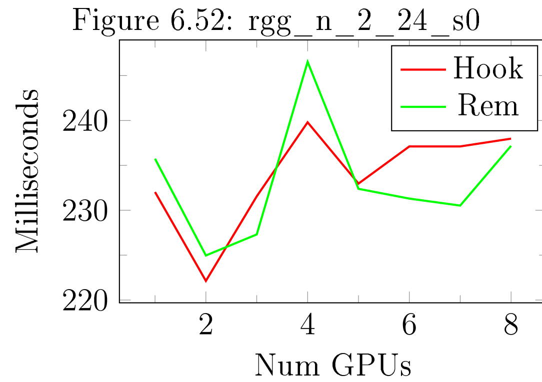


Figure 6.53: Run time performance on CUDA

6.4.1 Run time performance Hook vs Rem

Figure 6.50 and Figure 6.53 show the run time performance on four different graphs. Hook and Rem with on iteration of Hook-non-atomic was better on the asia.osm graph, this result is like how asia.osm ran on Groute, as street network graphs performed better with one iteration of Hook-non-atomic. Hook and Rem perform equally fast, but the run time increase with the number of GPUs for both. For the other three graphs are both Hook and Rem better with no iteration of Hook-non-atomic, for these graphs there is a run time performance improvement from one to two GPUs but perform worse from three and upwards. Hook and Rem varies between which is the best performing algorithm. In the coPapersDBLP Rem is consistently faster than Hook, though the difference is only 7% at one GPU and

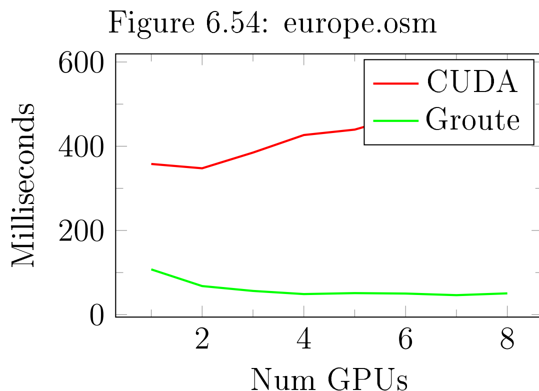


Figure 6.54: europe.osm

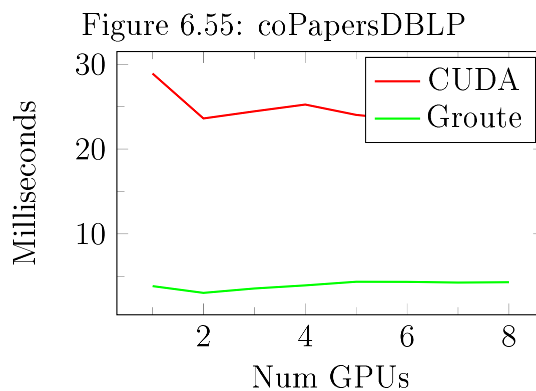


Figure 6.55: coPapersDBLP

Figure 6.56: Run time for our CUDA implementation and Groute

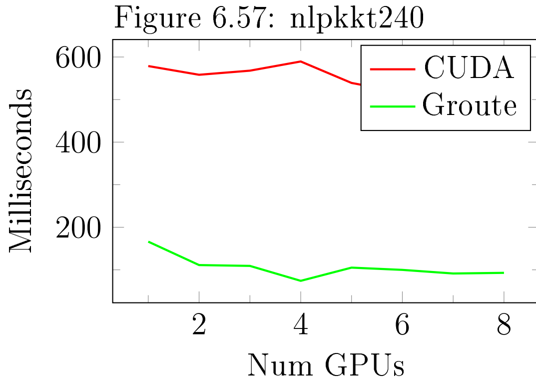
goes down to 1% difference at eight GPUs.

6.4.2 Compare run time between CUDA and Groute implementations

In this section show how our implementation of Rem in CUDA compare with our implementation of Rem in Groute. As Groute has taken steps to optimize memory handling and communication do we presume it will be faster as our CUDA implementation is not as optimized. Figure 6.56 and Figure x:G-cVG2 show this comparison on four graphs. In these four graphs we show that our CUDA implementation is much slower than our Groute implementation. With CUDA our run time increase as the number of devices are increase on the europ.osm and rgg_n_2_24_s0 graphs. At one GPU these graphs are between three and four times slower, at eight they are ten to eleven times slower. For nlpkkt240 and coPapersDBLP the CUDA implementation is more level and is constantly around three to six times slower on one to eight GPUs.

6.4.3 Strong scaling

In this section we show the strong scaling efficiency for the CUDA implementation on six graphs. With strong scaling we look at efficiency as the number of GPUs increase and the problem size stay the same. For all graphs the efficiency is better at two GPUs and decrease as the number is of GPUs increase. Even at two GPUs the efficiency is lower than what we would like, the CUDA implementation don't



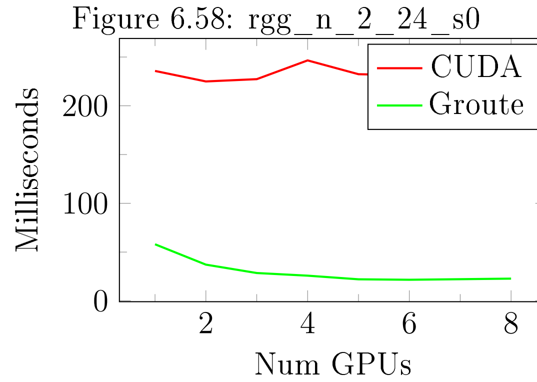
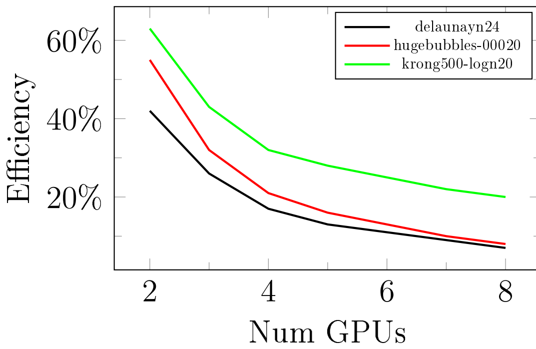


Figure 6.59: Run time for our CUDA implementation and Groute



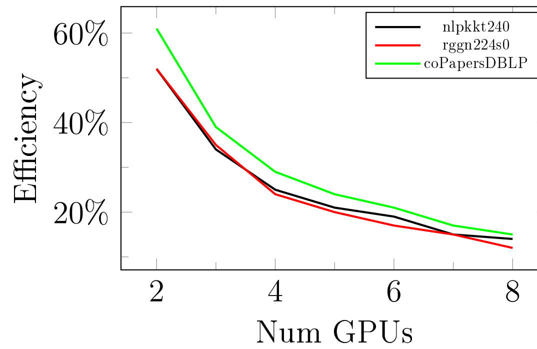


Figure 6.60: Strong scaling efficiency for the CUDA implementation

scale well.

6.4.4 Weak scaling

In this section we look at the weak scaling efficiency of our CUDA implementation of Rem. Our goal is to see how efficiently it scales when problem size is increased, and the number of devices increase. We use the rgg_n_2_22_s0 graph and the expanded versions as the scalable problem size, the graphs are presented in Section 6.1. Figure 6.42 shows the weak scaling efficiency for Rem and Hook. The goal is a high percentage close to 100%, an efficiency of 100% is unrealistic as we get more overhead work as the number of devices increases. The efficiency is not good at 60% running on two GPUs, and it drops further as the number of GPUs increase.

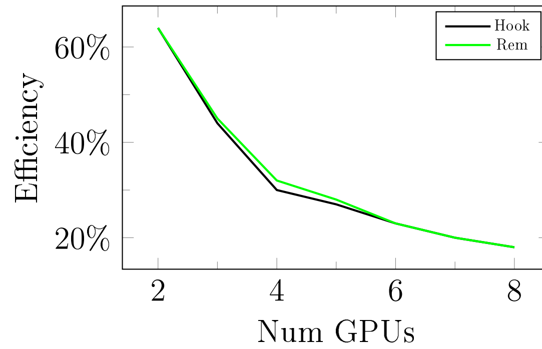


Figure 6.61: Weak scaling efficiency using rgg_n_2_22_s0 on CUDA implementation

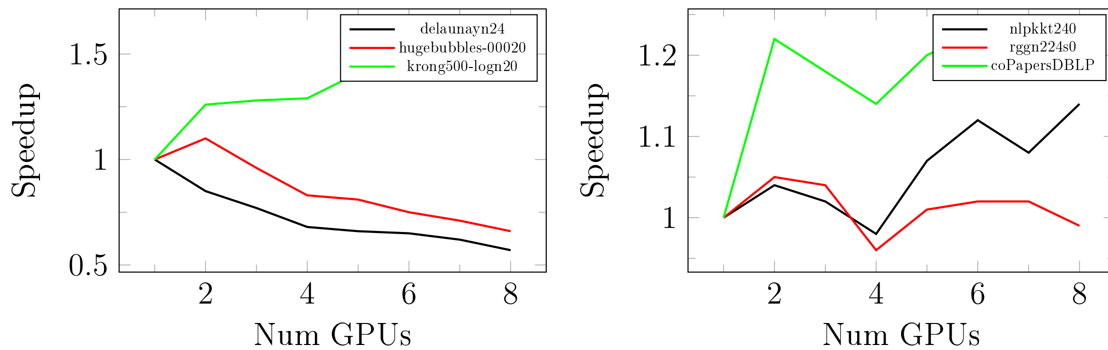


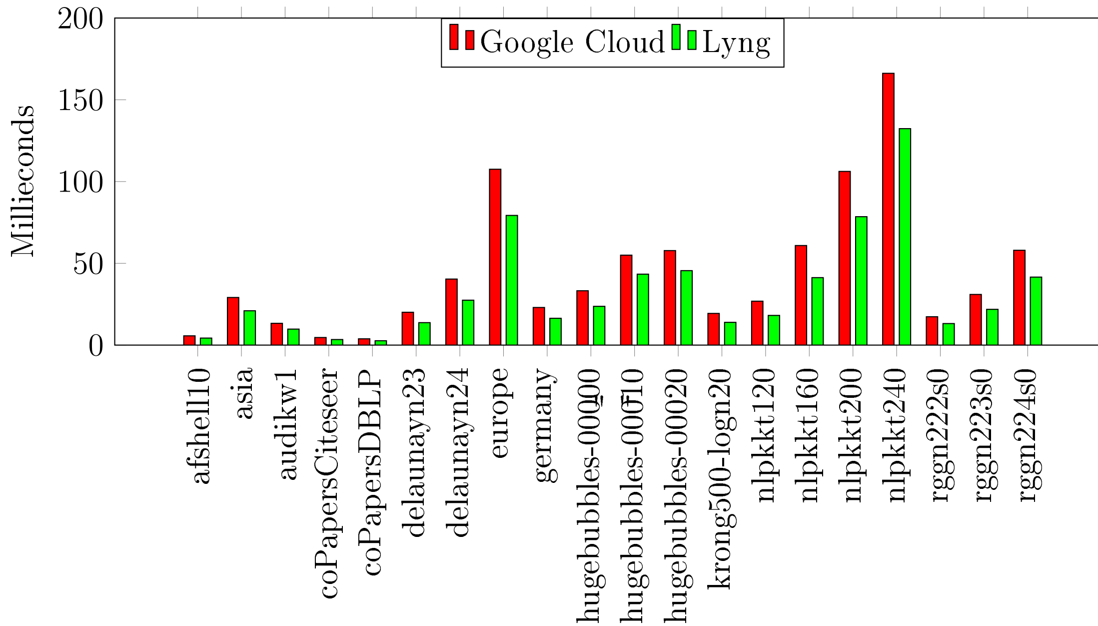
Figure 6.62: Speedup for the CUDA implementation

The CUDA implementation is not scaling as well as we would like when the input size and number of GPUs are.

6.4.5 Speedup

Figure 6.62 show the speedup for our CUDA implementation using Rem with no iteration of Hook-non-atomic running on six graphs. The speedup for the CUDA implementation is very low or none existing.

Figure 6.63: Comparison for Groute runnign Rem algorithm on Google Cloud and Lyng



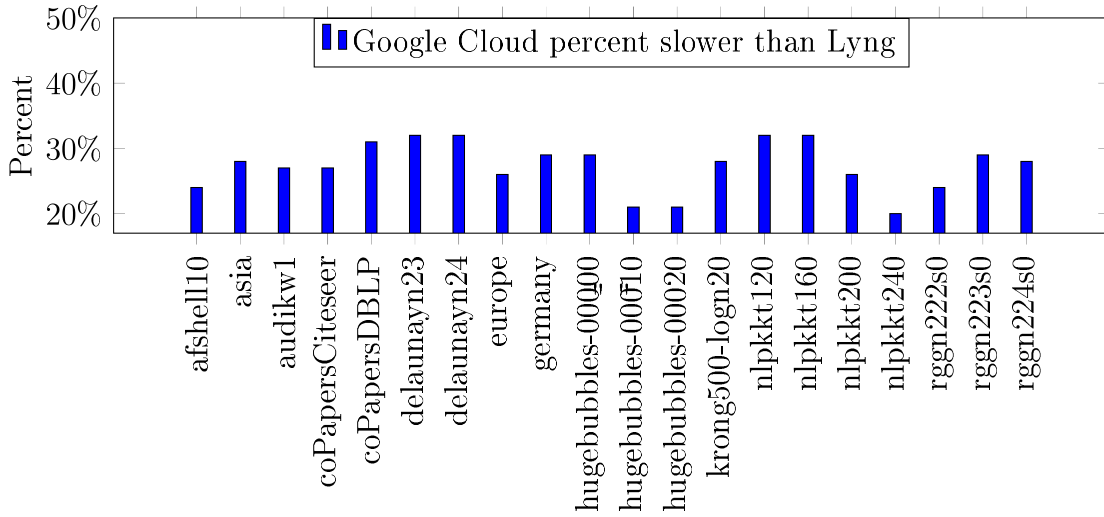
6.5 Distribution bottleneck Google Cloud

In this section we show how Google Cloud server and Lyng compares when Groute runs using one GPU. As Google Cloud has the better GPU we expect it to be faster than Lyng, both computers are described in Section 2.5. For these tests we use Groute running the Rem algorithm without Hook-non-atomic iterations as it is the best performing algorithm in Groute.

Figure 6.63 show the run time for each graph for both Google Cloud and Lyng. Lyng has the fastest run time of the two computers on all graphs. Figure 6.63 show the percentage Google Cloud is slower on each graph. Google Cloud is on average 27% slower than Lyng, range from 20% slower to 32% slower. Running the same test using Hook with an iteration of Hook-non-atomic yields similar results.

That Lyng is significantly faster with a slower GPU than Google Cloud opens a question if there is a bottleneck in the Google Cloud setup. Where the bottleneck exist in the setup is not certain, but if it affects the memory transfer it could hinder scaling when increasing the number of devices, as computer cant transfer data fast enough to all devices to take advantage of the added compute power.

Figure 6.64: Percentage Google Cloud is slower than Lyng running Groute with Rem



Chapter 7

Conclusion

We separate the conclusion into two parts. We go into our experience programming on multiple GPUs in Section 7.1. As there are interesting findings regarding the Rem algorithm [2] we used for testing we go into this in Section 7.2. We also discuss possible further work in Section 7.3.

7.1 Experience

We learned in Chapter 5 that creating optimized multiple GPUs programs is hard. Using environment or libraries that support in optimizing the hard task such as communication and memory handling seems reasonable as this is a hard part to gain improved performance. This will let us focus on the computation steps in the program and improve these. Though there is a need for documentations or instructions to make libraries and run time environment more accessible.

7.2 Rem and findings

We show in Chapter 6 that Rem's algorithm is a faster Connected components algorithm than the original Hook algorithm in Groute. The only exception are the street network graphs, running on these are both Hook and Rem performing equally well. Though Rem had generally a better run time was the scale efficiency and speedup not as good. This lack of scale efficiency and speedup could be because of the computer test setup, Google Cloud, described in Section 6.5.

Regarding the CUDA implementation is it slower and scale worse than Groute when we compare the same algorithm on the same graphs. The results from the

test with our CUDA implementation are not as interesting since the performance from this implementation is not good enough.

7.3 Further work

We show in Chapter 6 that Rem perform significantly better than the original Connected components in Groute. This performance difference is not as significant on our CUDA implementation, but as we show is the run time performance for our CUDA implementation much slower than the Groute implementation. The increased performance that Rem has, shows the potential of this algorithm in parallel computing. Further work could be to implement Rem in similar environments or libraries as Groute.

In Section 6.5 we open a question if there is a bottleneck on our Google Cloud test computer. It might be interesting to see if Groute running Rem would scale better on another setup for multiple GPUs.

We show that even Rem's algorithm see improvement using one iteration of Hook-non-atomic on street network graphs. Finding a reason why Rem see this improvement on these graphs and then find a possible improvement to Rem's algorithm.

Bibliography

[1] (2014). *Professional CUDA C Programming* (1st ed.). Birmingham, UK, UK: Wrox Press Ltd.

[2] (2018). *F. Manne, personal communications.*

[3] (2018, November). *https://cloud.google.com/.*

[4] (2018, March). *https://github.com/groute/groute.*

[5] (2018a, January). *https://github.com/gunrock/gunrock.*

[6] (2018b, October). *https://gunrock.github.io.*

[7] Bader, D. A., H. Meyerhenke, P. Sanders, and D. Wagner (2013). Graph partitioning and graph clustering, 10th dimacs implementation challenge workshop. *Contemporary Mathematics 588.*

[8] Ben-Nun, T., M. Sutton, S. Pai, and K. Pingali (2017). Groute: An asynchronous multi-gpu programming model for irregular computations. In *ACM SIGPLAN Notices*, Volume 52, pp. 235–248. ACM.

[9] Dasgupta, S., C. H. Papadimitriou, and U. Vazirani (2008). *Algorithms* (1 ed.). New York, NY, USA: McGraw-Hill, Inc.

[10] Geisberger, R., P. Sanders, and D. Schultes (2008). Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 90–100. Society for Industrial and Applied Mathematics.

[11] Holtgrewe, M., P. Sanders, and C. Schulz (2010). Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12. IEEE.

[12] Marquardt, O. and S. Schamberger (2005). Open benchmarks for load balancing heuristics in parallel adaptive finite element computations. In *PDPTA*, pp. 685–691.

[13] Pacheco, P. (2011). *An Introduction to Parallel Programming* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[14] Patwary, M. M. A., J. Blair, and F. Manne (2010). Experiments on union-find algorithms for the disjoint-set data structure. In *International Symposium on Experimental Algorithms*, pp. 411–423. Springer.

[15] Storti, D. and M. Yurtoglu (2015). *CUDA for Engineers: An Introduction to High-Performance Parallel Computing* (1st ed.). Addison-Wesley Professional.

[16] Wang, Y., A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens (2016). Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, Volume 51, pp. 11. ACM.