Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Vegard Seim Karstang

Multi-GPU Maximum Flow Using the Groute Framework

Master's thesis in Computer Science
Supervisor: Anne C. Elster
March 2020

NTNU
Norwegian University of
Science and Technology

Vegard Seim Karstang

Multi-GPU Maximum Flow Using the Groute Framework

Master's thesis in Computer Science
Supervisor: Anne C. Elster
March 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Project Description

Multi-GPU systems are becoming more common and powerful. This thesis focuses on how we can use these systems for graph problems. The specific problem this thesis focuses on is the maximum flow problem. The maximum flow problem is an important optimization problem with many uses. Solutions to the problem has been known for over 50 years, and over time several parallel solutions has been developed. Some of these have been implemented on a single GPU, but we are not aware of any multi-GPU version.

To aid in implementing a multi-GPU version we will use the Groute framework. This framework for asynchronous irregular processing on multiple GPUs was introduced in 2017 by Ben-Nun, Sutton, Pai, and Pingali. The goal of this project is to explore Groute, and to see whether the maximum flow problem is suited for multiple GPUs or not.

Abstract

The maximum flow problem is a core graph problem and algorithms for solving this problem has evolved since Ford and Fulkerson published their solution in 1956. It has applications in several graph flow problems, graph cuts, scheduling, and many more. These applications led to the development of preflow-push algorithms. These have several variations with different complexities, and are amenable to parallelization. Early efforts in parallelizing preflow-push algorithms needed locks to prevent errors, while recent research has led to lock-free algorithms that are much faster.

Multi-GPU systems are becoming more common and powerful. This has allowed the research into the usage and programming of such systems to increase. We introduce what is to our knowledge the first implementation of a maximum flow algorithm targeting multiple GPUs, and identify and discuss the suitability of implementing maximum flow on multiple GPUs.

Our implementation is based on the asynchronous multithreaded algorithm by Hong and He, and we use the asynchronous multi-GPU framework Groute by Ben-Nun, Sutton, Pai, and Pingali as a base. We use the distributed worklist supplied by Groute to manage our list of active items, and CUDA managed memory for host to device, and device to device communication. The implementation is benchmarked on an NVIDIA DGX-2 machine using genrmf graphs which are commonly used for benchmarking max-flow algorithms.

Our multi-GPU version may solve bigger problems compared to single GPU implementations due to the increased memory capacity. Even with this increase in memory capacity and the large amount of available threads it does not beat a simple single threaded CPU implementation. The implementation may also produce wrong results. We propose ways to combat both the run time issues, and the correctness issues.

We nevertheless managed to develop and implement a multi-GPU max flow algorithm which ran correctly on most of the benchmarks we tested. Our results illustrate how challenging implementing irregular algorithms efficiently on multiple GPUs.

Sammendrag

Maks flyt problemet er et kjerneproblem innen grafteori og algoritmer for å løse problemet har utviklet seg siden Ford og Fulkerson publiserte sin løsning i 1956. Dette problemet dukker opp i forskjellige flytproblemer i en graf, kutting av en graf, planlegging med flere. Disse bruksområdene første til det som kalles preflow-push algoritmer. Disse har flere varianter med varierende kompleksitet, og kan parallelliseres. De tidlige parallelle algoritmene brukte låser for å unngå feil, men nyere forskning har ført til algoritmer som ikke trenger låser og er derfor mye raskere.

Systemer med flere GPU-er er blitt mer vanlig og kraftigere. Dette har ført til en økning i forskning som både bruker, og utforsker hvordan man kan bruke slike systemer. Vi presenterer det som etter vår kunnskap er den første implementeringen av en maks flyt algoritme på flere GPU-er, og identifiserer og diskuterer utfordringer ved en slik implementasjon.

Vår versjon er basert på en asynkron flertrådet algoritme publisert av Hong og He. Vi bruker Groute som er et asynkront rammeverk for programmering av flere GPU-er. Groute ble publisert av Ben-Nun, Sutton, Pai og Pingali. I vår implementasjon bruker vi distribuerte arbeidslister som følger med Groute, og CUDA sitt innbyde minnehåndtering for kommunikasjon mellom enheter. Vi testet implementasjonen på en NVIDIA DGX-2 med genrmf grafer som er ofte brukt til å teste maks flyt algoritmer.

Fler-GPU versjonen vår kan løse større problemer enn andre versjoner som kun bruker en GPU ved å utnytte den økte minnekapasiteten. Problemet med vår versjon er at den har dårligere ytelse enn en enkel implementasjon på en vanlig prosessor selv med økt minnekapasitet. Implementasjonen er også ikke feilfri og kan produsere feil resultat. Vi foreslår hvordan man kan optimalisere implementasjonen for å redusere kjøretiden, og hvordan man kan hindre at man får feil resultat.

Selv med disse problemene klarte vi å utvikle og implementere en maks flyt algoritme som kjører på flere GPU-er og gir korrekt resultat på de fleste testene våre. Resultatene våre viser hvor vanskelig det å implementere uregulære algoritmer effektivt på flere GPU-er.

Aknowledgements

I would like to thank Dr. Anne C. Elster for supervising this work. Her advice has been invaluable, and her patience with me while I take more time than normal to finish is greatly appreciated.

I would also like to thank her and all the other members of the HPC-Lab for making it a great place to work. Thank you to NTNU, NVIDIA, IBM, and other sponsors who have supported the lab by supplying or loaning equipment, or in other ways helped the lab and its members.

In the same vein I would like to thank all the other staff at NTNU that in some way supports the lab and the support staff for the DGX-2 that is used in this work.

Finally, a thank you goes to the ISS group at the Univeristy of Texas at Austin, including Professor Pingali, Dr. Jatala, and Mr. Hoang for answering question about Groute, and for their hospitality during my visit in 2018.

Table of Contents

List of Tables

List of Figures

Listings

List of Algorithms

Abbreviations

BSP	=	Bulk Synchronous Parallel
CPU	=	Central Processing Unit
CSC	=	Compressed Sparse Column
CSR	=	Compressed Sparse Row
DDR	=	Double Data Rate
ECC	=	Error-Correcting Codes
FLOPS	=	Floating Point Operations Per Second
GPGPU	=	General Purpose computing on Graphics Processing Units
GPU	=	Graphics Processing Unit
HPC-Lab	=	Heterogeneous and Parallel Computing Laboratory at The Department of Computer Science, NTNU
HPC	=	High Performance Computing
MSB	=	Multilevel Spectral Bisection
RAM	=	Random Access Memory
SIMT	=	Single-Instruction Multiple-Thread

Chapter 1

Introduction

The high performance computing community is always demanding more processing power, and the manufacturers have risen to the challenge by creating accelerators external to the processor that the processor may take advantage of. The most prevalent of these accelerators is the Graphics Processing Unit (GPU). The GPU provides the processor with a large amount of parallel processing power and the memory bandwidth capable of handling large amounts of data. The threads in a GPU execute in a single-instruction multiple-thread (SIMT) manner, and the executing threads must cooperate in order to fully benefit from the GPU hardware. This cooperation requirement makes implementing irregular and sparse algorithms on the GPU difficult. However it is still possible to achieve speedups over CPU-based implementations for these algorithms.

Researchers and software developers are increasingly targeting multi-GPU system as their availability and capability grows. Programming and utilising the resources a single GPU gives you is challenging, and adding more GPUs does not make it any easier. These difficulties are more prominent when implementing irregular algorithms such as many graph algorithms. The maximum flow algorithm we are implementing is one example of an irregular algorithm.

Groute [4] is a framework for programming multi-GPU system. The framework targets asynchronous, irregular applications, and provides several constructs to ease programming. Some of these as low level communication constructs that use packetization, while others are more high level. One of the high level constructs provided is the distributed worklist which eases development of applications.

Maximum flow is a graph algorithm which is used in many graph flow problems as well as graph cuts, scheduling, optimization and more [2]. Many of these problems require a performant max flow algorithm, and a lot of work has been put into improving and parallelizing max flow algorithms since Ford and Fulkerson published their original solution in 1956 [5]. Recent research improves the old lock based parallel algorithms by using atomic instructions to remove any locks [3]. Implementing max flow algorithms on a single GPU has been done before by Gunrock [6] and others.

1.1 Goals and Contributions

This thesis describes how we implement the first multi-GPU maximum flow algorithm based on previous research in parallelizing preflow-push algorithms. The goal of this research is to find the suitability of preflow-push algorithms on multiple GPUs. The way we test this is using genrmf graphs which are synthetic graphs used for testing max flow implementations. The graph is partitioned over the GPUs and we measure wall clock time and error rates. Based on the benchmarking and time spent during implementation we identify challenges that arise when implementing and solving max flow on multiple GPUs. We also propose ways to approach these challenges.

1.2 Outline

The rest of this report is structured as follows:

- In Chapter 2, we provide background information about GPU programming, graphs, and the maximum flow problem.
- In Chapter 3, we present information about our implementation.
- In Chapter 4, we benchmark our implementation, present and discuss the results.
- In Chapter 5, we provide suggestions for future work and conclude.
- In Appendix A, we provide additional CPU benchmarks and link to the CPU source code.
- In Appendix B, the main part of the source code we developed is listed as well as a link to the full source code is provided.

Chapter 2

Background

This chapter gives an overview of the main background information about GPUs, GPU programming, graphs, and the maximum flow problem needed for the rest of the thesis. Note that the sections about GPUs and graphs are adapted from the author's specialization project [7].

2.1 The Graphics Processing Unit

As the name suggests the *Graphics Processing Unit* (GPU) was originally designed as a graphics processor. GPUs can render complex 3D visualisations by creating a thread for each node and pixel fragment and run them in parallel [8]. Using this highly parallel architecture, programmers started writing simulations and other more general purpose programs for the GPU. These general purpose GPU (GPGPU) computations achieved high performance, but programmers struggled to write them because they had to express non-graphics calculations using graphics APIs.

To solve this problem, NVIDIA introduced the GeForce 8800 in 2006 [8]. It is the first GPU where the graphics and compute architecture is unified and programmable using C and the new CUDA programming model for general computation while keeping OpenGL and DirectX for graphics.

Todays GPUs support GPGPU programming extensively with IEEE 754 floating-point operations, cached memory with ECC memory protection, and lately also support for tensor cores [1].

2.1.1 The NVIDIA Volta GPU architecture

The NVIDIA Volta GPU architecture was released by NVIDIA in 2017 and is one of the latest GPU architectures from NVIDIA [1]. The high end V100 GPU was released along with the reveal of the new architecture. An overview of the architecture of the V100 is shown in figure 2.1. The V100 has 80 streaming multiprocessors each with 64 CUDA cores for a total of 5120 total cores. With these the SXM2 version of the Tesla V100 can reach 7.5 TFLOPS of double-precision performance, and 15 TFLOPS for single-precision [9]. The streaming multiprocessors also come with 8 tensor cores for a total of 640 tensor cores. These cores are used to accelerate deep learning applications, and can deliver up to 120 TFLOPS of performance [1].

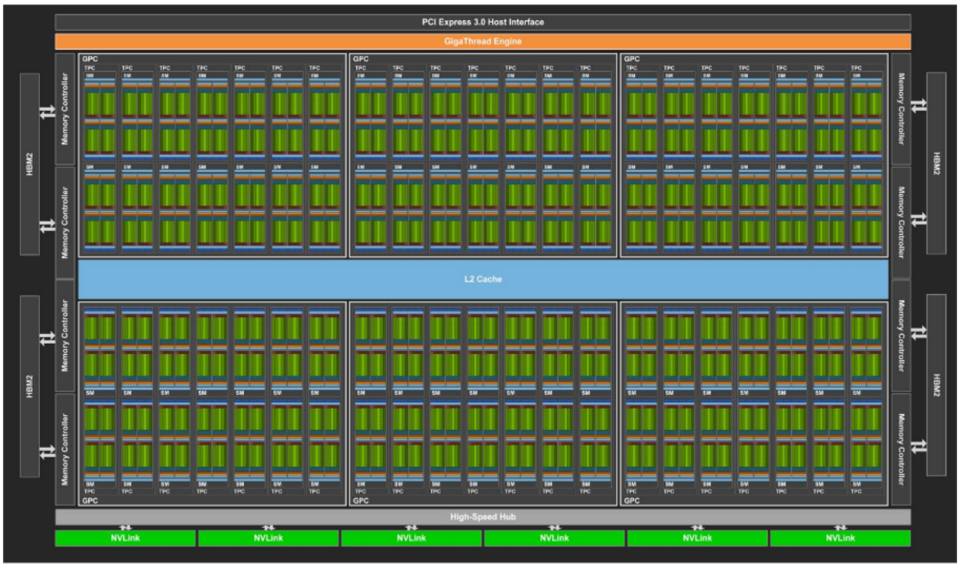

Figure 2.1: Overview of the NVIDIA Volta V100 GPU [1]. Note that this shows a full GV100 GPU with 84 streaming multiprocessors, and that the V100 mentioned and used in this work has 80 streaming multiprocessors available. With permission from NVIDIA.

Application execution

During application execution, the Volta can start up to 2048 threads per streaming multiprocessor. These threads are managed by splitting them into warps containing 32 threads each. These threads are executed in a single-instruction multiple-thread (SIMT) manner. In earlier GPU architectures from NVIDIA all 32 threads shared a common program counter and stack while in the new Volta architecture each thread has its own program counter and stack. This allows more complex and fine-grained algorithms while the new convergence optimizer keeps the efficiency high. This optimizer is necessary because excessive amounts of branch divergence hinders performance.

When threads in a warp accesses memory addresses the memory system tries to coalesce these accesses to reduce the amount of actual memory accesses. An example of this is that if all 32 threads in a warp accesses memory within a single 128-byte segment, the accesses can be coalesced into a single memory access. If the memory system can't coalesce accesses the memory is read in a serial manner which degrades application performance.

2.1.2 NVLink and NVSwitch

NVLink is NVIDIA's own interconnect for GPU-to-GPU and CPU-to-GPU communication [10]. The first version was released together with the Pascal GPU architecture in 2016, and the current version was released alongside the Volta GPU architecture [1]. The current version of NVLink supports 12 sub-links, and two sub-links form a bidirectional link between two processors. Each link provides 25 GB/s communication in each direction for a total of 50 GB/s bidirectional bandwidth. The maximum bandwidth over all six links is 300 GB/s. NVLink gives the CPU direct load/store/atomics access to each of the connected GPU's memory, and it supports atomics that are initiated by either the CPU or the GPU. NVLink also has other features such as a low-power mode when a link is not heavily used, and it allows GPUs to directly access the CPU's page tables through its address translation services.

The NVSwitch is an NVLink based switch with 18 NVLink ports per switch [11]. The switch has a fully connected, non-blocking internal crossbar which allows all ports to communicate with all other ports without any loss of bandwidth. Cyclical redundancy coding (CRC) is used to detect any transfer error over the NVLink, and to replay any communication if necessary. Error-correcting codes (ECC) protects the datapaths, state-structures, and routing. Other security measures include buffer under/overflow checks, and indexed routing tables that are managed by the fabric manager to prevent applications from accessing memory outside their specific ranges.

NVLink and NVSwitch is heavily used in NVIDIA's DGX-2. The DGX-2 contains 2 baseboards with 6 NVSwitches and 8 GPUs each for a total of 12 NVSwitches and 16 GPUs [11]. Each GPU on a baseboard is connected to all switches on the same baseboard, and the baseboards are connected using 8 ports on each of the switches. This topology allows the GPUs to communicate at 300 GB/s, and allows the GPUs to behave as one from the outside.

2.1.3 Multi-GPU Nodes

Multi-GPU nodes are now quite common in systems designed for high-performance computing. These nodes consist of one or several hosts (CPUs) connected to several GPUs via a low-latency, high-throughput interconnect such as PCI-Express or NVLink [4]. The CPUs and GPUs can be interconnected in several different topologies. Examples of such topologies are the tree-topology and the all-to-all topology. The DGX-2 uses an all-to-all topology [11]. The goal of these multi-GPU nodes is to allow parallel applications to take advantage of the extra memory and computational power of multiple GPUs.

The are two main methods used to program multiple GPUs [4]. The simplest method is to manage each GPU separately by using a single process per GPU. The other method is the Bulk Synchronous Parallel (BSP) method. An application using the BSP method executes computations in rounds followed by global communication. The computations are done locally on each GPU.

There are problems with both of these methods that may result in underutilization of the GPUs [4]. The BSP method does global synchronization between rounds which can result in unnecessary serialization. Managing each GPU separately requires some kind of message-passing interface for communication which increases the overhead of the CPU part of the program.

One way to minimize the problem of underutilization of GPUs is to use an asynchronous method [4]. An asynchronous method allows each GPU to compute and communicate with other devices independently of other GPUs or CPUs. The main difficulty with this is that developing asynchronous applications is challenging, and requires an in-depth knowledge of both code, and the underlying architecture and communication network the code runs on.

2.1.4 The CUDA Programming Model

As mentioned earlier, the CUDA programming model was introduced in 2006 by NVIDIA together with the GeForce 8800 GPU [12] [8]. Initially CUDA was an extension to the C programming language, and over time more languages has started supporting CUDA either through language extensions or by calling C/C++ code. CUDA is designed to enable development of highly parallel programs that can run using the thousands of threads available on a GPU. CUDA abstracts away much of the complexity of the GPU which allows a single CUDA program to execute on GPUs of varying sizes [8].

Programming with CUDA is similar to programming in C/C++. The only difference is that the code also contains *parallel kernels* that are run in parallel on the GPU. Kernels are regular code that is compiled to GPU instructions and run when the serial part of the program calls them using a specialized function call syntax.

CUDA organizes threads into a grid of thread blocks. Threads within a thread block can communicate using high-speed shared memory, and they can synchronize with each other via barriers. Communication between thread blocks is done via the global shared memory available on the GPU [8]. A single thread often uses information about the grid dimension, thread block, and its position within the block to figure out which part of the calculation it is supposed to do.

While thread blocks and grids sound similar to warps and simultaneous multiprocessors they are only an abstraction that allows a CUDA program to scale between different GPUs.

NVIDIA is always improving the CUDA memory model. Later versions have several new features one of which is unified memory which is important for our work.

Unified Memory

Another problem with programming multi-GPU nodes is memory management. Each GPU needs access to the memory it is operating on, which for multi GPU work means that memory has to be partitioned between the GPUs. CUDA solves this problem using unified memory. Unified memory was first introduced in CUDA 6 [10]. Unified memory gives the programmer a pool of managed memory that can be accessed from both the CPU and any GPUs using a single pointer. Moving the actual memory between the CPU and a GPU, or between GPUs is done by the CUDA system software.

Listing 2.1 shows how allocate a large vector using unified memory, initialize it on the CPU, and finally run a kernel on 2 GPUs. The most important takeaway from this is that there are no explicit calls that allocate or copy memory on the GPUs. We allocate our vector using unified memory, and CUDA handles memory movement for us.

Listing 2.1: Running a kernel on 2 GPUs with unified memory

```
int N = 1 << 20;
int * vec;
cudaMallocManaged(vec, N * sizeof(int));
for (int i = 0; i < N; ++i) vec[i] = i;
cudaSetDevice(0);
myKernel<<<...,...>>>(vec, N);
cudaSetDevice(1);
myKernel<<<...,...>>>(vec, N);
```

2.2 Graphs

A graph consists of a set of vertices connected by a set of edges. Formally this is written $G = (V, E)$ where V is the set of vertices and E is the set of edges connecting them. A single edge connects either two vertices or it connects a node with itself. Edges that connect a node with itself is called a *self-edge*. It is not unusual for edges or vertices to have additional information associated with them. For edges the most common example is a numeric value denoting some kind of weight or price associated with traversing that particular edge. Node information may be an identifier, some kind of value associated with a computation such as the shortest known path to that node from a starting node, or other needed values.

A graph containing weighted edges is called a weighted graph, while graphs without edge weights are called unweighted graphs. In a directed graph an edge can only be used to travel from one node to the other connected node. In an undirected graph all edges are symmetrical and one edge can be used to travel in both directions between the connected vertices.

The *degree* of a node is the amount of edges connected to that node. In a directed graph one may talk about the *in-degree* and *out-degree* of a node.

Graphs can be used to model many data sets that stem from real-world applications. Examples of such real-world applications are networks such as computer, social, road, and neural networks. Other examples are circuit simulation and protein interactions.

2.2.1 Real-world Graphs

Graphs gathered from real-world applications are often quite complex and without any simple structure. These graphs are often sparse and scale-free.

A sparse graph is a graph where $|E| \lll |V|^2$ [13]. The sparsity may be utilised in computer applications to reduce the memory footprint. This is done by using special sparse graph representations as discussed in section 2.2.3.

A scale-free graph or network is a network where the degree of each node follow a power law distribution of $P(k) \sim k^{-\gamma}$ [14]. Examples of such graphs are links on internet pages, protein interactions, and social networks. The result of this skewed degree distribution is that some vertices have a much larger degree compared to most other vertices.

This may cause load-balancing problems when processing the graph on the GPU resulting in lower performance.

2.2.2 Graph Representations

Representing a graph on a computer can be done in several different ways. One way is using $|V|$ *adjacency-lists*. An adjacency list is a list of vertices $v \in V$ such that for a node u the edge (u, v) exists. The weight is stored together with the node if the graph is weighted. This is a quite robust and modifiable representation requiring only $\Theta(V + E)$ space, but it comes short if one needs to know if a certain edge exists in the graph [13].

Another representation is the *adjacency-matrix* representation. An adjacency-matrix, A, on a graph with $|V|$ vertices is a $|V| \times |V|$ matrix, where the entry A_{ij} represents the edge from i to j with $i, j \in V$. In a weighted graph the element can represent the weight on the edge, while in an unweighted graph the element is usually 0 or 1 to indicate if an edge exists or not. In an undirected graph we have that $A = A^T$ because (u, v) and (v, u) represent the same edge. In this case all elements below the diagonal does not need to be stored which saves memory. One benefit of this representation is that checking for the existence of, or getting information from an edge is a simple look-up in the matrix.

For sparse graphs the adjacency-list approach is more memory efficient due to it only needing $\Theta(V + E)$ compared to the $\Theta(|V|^2)$ an adjacency-matrix needs. More specialized schemes for sparse graph representation is discussed in section 2.2.3.

2.2.3 Sparse Graph Representations

Several storage schemes have been proposed to take advantage of the amount of zero elements in large sparse matrices. These matrices often show up when one is trying to represent real-world graphs and problems. The main goal of these schemes is that they allow for common matrix operations while only storing the nonzero elements of the matrix [15].

In the following subsections m is the width and height of the corresponding sparse matrix, and nz is the number of non-zero elements in the matrix.

Coordinate format

The coordinate format consist of three arrays. One containing the row indices, another containing the column indices, and the last containing the corresponding value. All three arrays are of length m [15].

CSR and CSC

Compressed Sparse Row (CSR) is one of the most used formats for storing sparse matrices [15]. Like the coordinate format it consist of three arrays. The first array, AA, contains the non-zero values stored row by row. The second array, JA, contains the column indices of the values stored in AA. The last array, JA, contains pointers to the start of each row in AA and JA. The total space needed for the first two arrays is $2m$ while the last array only needs nz space. Compressed Sparse Column (CSC) is the same as CSR except that everything is done in a column-major order instead of the row-major order of CSR [15]. There are also quite a few similarities between CSR and the coordinate format.

2.2.4 Graph Partitioning

The graph partitioning problem has an extensive amount of applications in areas of scientific computing, task scheduling, and more [16]. We use graph partitioning algorithms to partition the graph such that each GPU gets its own part of the graph to work on.

Formally, the *k-way graph partitioning problem* is the problem where given a graph $G = (V, E)$, we wish to partition V into k subsets $V_1, V_2, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = \frac{n}{k}$, $\bigcup_i V_i = V$, and the number of edges in E whose incident vertices belong to different subsets is minimized [16].

Finding the optimal solution to this problem is NP-complete. This does not mean that we can't find a reasonably good partition in a reasonable amount of time however, and several methods have been developed that does this. Some of the main graph partitioning methods are spectral partitioning, geometric partitioning, and multilevel partitioning.

Spectral partitioning methods are expensive to compute because they require the eigenvector corresponding to the second smallest eigenvector of the Laplacian matrix of the graph [16]. Despite this they are often used because they produce good partitions for a large amount of problems. Multilevel Spectral Bisection (MSB) is one of the fastest algorithms spectral algorithms and combines the spectral and multilevel methods.

Geometric partitioning methods may be faster than spectral methods, but they often create partitions that are worse than the ones created by a spectral method. This method is also limited in usage because it can only be used if vertex coordinates are available.

Multilevel partitioning methods work by coarsening, partitioning, and then uncoarsening the graph. The coarsening is done by transforming the initial graph G_0 to a much smaller graph G_m in a sequence of transformations $G_1, G_2, \ldots, G_m$ where $|V_0| > |V_1| > \cdots > |V_m|$. After coarsening a partition P_m of the smaller graph $G_m = (V_m, E_m)$ is computed with a partitioning algorithm. The uncoarsening is done by going through the graphs $G_{m-1}, G_{m-2}, \ldots G_1$. At each step i the partition P_i is refined because while P_{i+1} is a local minimum partitioning of G_{i+1}, the projected partition P_i may not be a minimum local partitioning of G_i.

2.3 The Maximum Flow Problem

The maximum flow problem is one of many network flow problems [2]. Network flow problems are problems where we wish to move some entity such as electricity, a product or a car from one point to another through a network as efficiently as possible. In the maximum flow problem the network is modelled as a directed weighted graph and we wish to move as much of some entity as possible from the source node s to the sink node t. This has to be done without breaking the maximum capacity of any of the edges in the graph.

If x_{ij} represents the current flow over the edge from node i to node j, and c_{ij} represents the maximum capacity of the edge the problem can be stated more formally as a linear programming problem as follows [2].

Maximize v

Subject to

$$\sum_{\{j:(i,j)\in E\}} x_{ij} - \sum_{\{j:(j,i)\in E\}} x_{ji} = \begin{cases} v & \text{for } i = s \\ 0 & \text{for all } i \in V - \{s,t\} \\ -v & \text{for } i = t \end{cases}$$

$$0 \le x_{ij} \le c_{ij} \text{ for each } (i,j) \in E$$

The main constraint in the program is the mass balance constraint that maintains equal amounts of flow in and out of a node that is not the source or sink. The second constraint is the flow bound constraint that keeps the amount of flow over an edge under the capacity of the edge.

There are several algorithms that solve the maximum flow problem in polynomial time [2]. Examples are the capacity scaling algorithm, the shortest augmenting path algorithm, and preflow-push algorithms.

For the rest of this section $G.E$ and $G.V$ refers to the set of edges and nodes in the original network, and $G(f).E$ and $G(f).V$ refers to the edges and nodes in the residual network with respect to the flow f.

2.3.1 Flows

A flow in a graph G is a function $f : V \times V \to \mathbb{R}$ that for each edge in G satisfies the mass balance and flow bound constraint. The value $|f|$ of a flow is defined as [13]

$$|f| = \sum_{i\in G.V} x_{si} - \sum_{i\in G.V} x_{is}$$

where s is the source node.

Preflows

A preflow is a flow that satisfies the flow bound constraint, and the following relaxation of the mass balance constraint [2].

$$\sum_{\{j:(i,j)\in G(f).E\}} x_{ij} - \sum_{\{j:(j,i)\in G(f).E\}} x_{ji} \ge 0 \quad \text{for all } i \in G(f).V - \{s,t\}$$

This relaxation allows nodes to have excess flow, and for a given preflow we define the excess flow at a given node i as:

$$e(i) = \sum_{\{j:(i,j)\in G(f).E\}} x_{ij} - \sum_{\{j:(j,i)\in G(f).E\}} x_{ji}.$$

For any given preflow we have that $e(i) \ge 0$ for all $i \in G(f).V - \{s\}$. The source node is the only node that can have negative excess, and always has negative excess in any preflow.

2.3.2 The Residual Network

The residual network plays an important part when describing and developing algorithms for the maximum flow problem [2]. The residual network arises from the original network we are solving the maximum flow problem in. In addition to the edges in the original network, the residual network has the edge (j, i) whenever (i, j) is an edge in the original network. If the edge (j, i) already exists then the capacity of this edge is changed to the sum of the capacities of the edges (i, j) and (j, i). Using this we can define the *residual capacity* r_{ij} as follows:

Given a residual network, the residual capacity of an edge $(i, j) \in G(f).E$ is maximum additional amount of flow that can be sent from i to j using both (i, j) and (j, i). The residual capacity of an edge (i, j) is the sum of the unused capacity of the edge and the current flow on (j, i) which can be used to cancel flow on (i, j). Mathematically this is written $r_{ij} = u_{ij} - x_{ij} + x_{ji}$.

Given a flow f, the only edges in the residual network $G(f)$ are the edges with positive residual capacity with respect to the flow f [2].

2.3.3 Distance Labels

Distance labels are nonnegative integer labels given to the nodes in the residual network [2]. The distance labels arise from a *distance function* $d : G(f).V \to \mathbb{Z}^+ \cup \{0\}$ which is a function from the set of nodes in the residual network to the nonnegative integers. A distance function is said to be valid if it satisfies the following two conditions for a given flow f:

$$\begin{aligned} &d(t) = 0 \\ &d(i) \leq d(j) + 1 \quad \text{for every edge } (i, j) \text{ in the residual network } G(f) \end{aligned}$$

The value of $d(i)$ is the distance label of node i.

The distance function has a couple of nice properties. The first one is that if the distance labels are valid then $d(i)$ is a lower bound on the length of the shortest path from node i to the sink [2]. The other property is that if $d(s) \geq |G(f).V|$, the residual network contains no directed path from the source to the sink.

Finally, any edge (i, j) where $d(i) = d(j) + 1$ is said to be an *admissible* edge.

2.3.4 Preflow-Push Algorithms

Preflow-Push algorithms are based on a generic algorithm for solving the maximum flow problem [2]. The algorithm has two main phases.

The first phase is the preprocess phase where the distance labels and initial preflow is computed. The initial preflow is computed by pushing as much flow as possible from the source to its neighbours. The distance labels are computed by running breadth first search from the sink. The initial preflow saturates all edges incident to the source which in turn removes any directed paths from the source to the sink in the residual network. This allows us to set the distance label of the source to $|V|$ [2].

The second phase is the push/relabel phase. In this phase the algorithm works on *active nodes*. A node i is active if its excess, $e(i)$, is greater than 0. We also adopt the convention that the source and the sink is never active. The source is never active because $e(s) \leq 0$ after the initial preflow. The sink is never active because pushing flow from the sink to any of its neighbours would only result in the neighbour pushing the flow back to the sink. This is wasted computation so the sink is never active. With this in mind the algorithm chooses an active node, and either pushes as much flow as possible from this node to one of its neighbours over an admissible edge or relabels it because there are no admissible edges in the residual network [2]. The pseudocode for this is found in Algorithm 1.

This generic algorithm has $O(|V|^2|E|)$ complexity [2]. There are several ways to implement this algorithm, and the main way they differ is how they choose the active node to process.

- The FIFO preflow-push algorithm processes active nodes in a first-in, first-out order. This algorithm has $O(|V|^3)$ complexity [2].
- The highest-label preflow-push algorithm chooses the active node with the highest distance label and pushes from this node. This algorithm has complexity $O(|V|^2\sqrt{E})$ [2].
- It is also possible to process all active nodes in parallel. This must be done carefully to prevent any data races when flow is pushed from several nodes to a single node in parallel. One of the first parallel max flow algorithms was published by Shiloach and Vishkin in 1982 [17].

In the next chapter we will develop a multi-GPU version based on this generic preflow-push algorithm.

Algorithm 1 The Preflow-Push algorithm [2]

```
1: procedure PREPROCESS
2:     $x = 0$
3:     compute the exact distance labels $d(i)$
4:     $x_{sj} = u_{sj}$ for each edge $(s, j) \in E$
5:     $d(s) = |V|$
6: end procedure
7:
8: procedure PUSH/RELABEL($i$)
9:     if the network has an admissible edge $(i, j)$ then
10:        push $\delta$ = $min\{e(i), r_{ij}\}$ units of flow from node $i$ to node $j$
11:    else
12:        replace $d(i)$ by $min\{d(j) + 1 : (i, j) \in E$ and $r_{ij} > 0\}$
13:    end if
14: end procedure
15:
16: algorithm PREFLOW-PUSH
17:    preprocess()
18:    while the network contains an active node do
19:        Select an active node $i$
20:        push/relabel($i$)
21:    end while
22: end algorithm
```

Chapter 3

Multi-GPU Max Flow

In this chapter we describe how we develop and implement the asynchronous parallel max-flow algorithm by Hong and He [3] on multiple GPUs using a framework called Groute [4] to handle distributed worklist creation and maintenance, and kernel launches on multiple GPUs. CUDA managed memory is used for general memory management and communication. The section about Groute describes some of its internals and programming model. We pay special attention the the distributed worklists provided by Groute. After this we describe the algorithm we are implementing, the global relabeling heuristic, and our attempts at implementing this heuristic.

3.1 Groute

Groute [4] is an asynchronous programming model and implementation. It targets irregular computations on multi GPU nodes. The implementation has constructs, common collective operations, and distributed work-lists that allow for easier implementation of irregular algorithms on multi GPU nodes. Implementing algorithms in Groute is done by first creating a dataflow graph of computation. This is done by connecting *endpoints* which are either physical devices, or *routers* which are used for communication of data. Endpoints are connected with *links* as long as there are no self loops. Communication over links are implemented with special *Send* and *Receive* methods. The asynchronous computation is done by creating workers that launch kernels on the GPUs and call the send and receieve methods for communication.

3.1.1 Distributed Worklists

The distributed worklist is a high level interface provided by Groute. The worklist contains a global list of items to be processed, and each item processed may create new items. Doing this in a distributed fashion requires all-to-all communication which is implemented using routers. The management of the worklist is centralized on the host. The host receives updates about the amount of items generated and processed, and ends the computation when there are no more items to process.

The worklist uses a single system-wide router for communication, and local worklists for any local items. For communication the worklist creates a ring topology of the GPUs and passes items around this ring until it reaches the correct GPU. To do this the worklist requires the programmer to implement five callbacks, *Pack*, *Unpack*, *OnSend*, *OnReceive*, and *GetPrio*. The *Pack* and *Unpack* callbacks are used to pack an outgoing item and unpack an incoming item respectively. The *OnSend* and *OnReceive* callbacks are used to determine the destination of outgoing and incoming items. Finally the *GetPrio* callback is used to obtain the priority of the incoming item.

3.2 Asynchronous Parallel Max-Flow

We implemented the algorithm by Hong and He found in [3] using Groute to target multiple GPUs. The algorithm is an asynchronous parallel max-flow algorithm using puhs-relabel. The main advantages of this algorithm is that it does not need locks or barriers during parallel execution. The algorithm requires $O(|V|^2|E|)$ operations to compute the maximum flow. There are a few key differences between this algorithm and the generic push-relabel algorithm. The main difference is that the active nodes are processed in parallel. The algorithm uses atomic operations to push flow through the network to prevent any data races, or other conditions that may cause the algorithm to produce the wrong result. Another change is that the algorithm does not push flow over an admissible edge. Instead it pushes flow to the lowest neighbour. This does not affect the correctness of the algorithm.

Pseudocode for processing a single node can be found in Algorithm 2. In this pseudocode e', v', h', h'' and Δ are per-thread private variables, u is the node being processed, $d(u)$ is the distance label of u, excess(u) is the excess at u, and $c_f(u, v)$ is the capacity of the edge in the residual network from u to v. The variables $d(u)$, excess(u), and $c_f(u, v)$ are shared among all threads.

Our implementation uses the distributed worklist found in Groute to manage the list of active nodes, and CUDA managed memory to handle all memory operations such as allocation and device to device communication. The graph is partitioned sequentially using the built in partitioner. We use the special global atomic functions found in CUDA to solve any issues when updating edges that span across partitions.

Algorithm 2 Pseudocode for node processing in the asynchronous max flow algorithm [3]

```
1:  Function PROCESS
2:      while excess(source) + excess(sink) $< 0$ do
3:          if excess($u$) $> 0$ then
4:              $e' =$ excess($u$)
5:              $h' = \infty$
6:              for all neighbours $v$ of $u$ in the residual network do
7:                  $h'' = d(v)$
8:                  if $h'' < h'$ then
9:                      $v' = v$
10:                     $h' = h''$
11:                 end if
12:             end for
13:             if $d(u) > h'$ then
14:                 $\Delta = min(e', c_f(u, v'))$
15:                 $c_f(u, v') = c_f(u, v') - \Delta$
16:                 $c_f(v', u) = c_f(v', u) + \Delta$
17:                 excess($u$) = excess($u$) - $\Delta$
18:                 excess($v'$) = excess($v'$) + $\Delta$
19:             else
20:                 $d(u) = h' + 1$
21:             end if
22:         end if
23:     end while
24: end Function
```

3.2.1 Global Relabeling

An effective way to reduce the runtime of max-flow algorithms is to augment it with a global relabeling heuristic [18]. Global relabeling is done by first doing a breadth-first search through the residual network from the sink. All nodes reached this way has their distance label updated to the distance found from the sink. Then the distance label of the source node is set to $|V|$, and breadth-first search is done through the residual network from the source. The nodes reached this way has their distance labels updated to $|V|$ plus the distance from the source.

We attempted to implement global relabeling in two different ways. Both of these attempts tried to do global relabeling on the CPU which may be the source of the problems that was encountered. The first attempt took an asynchronous approach. We created an extra CPU thread to run the global relabeling operation, and set it to run every $|V|/2$ iterations. We also changed the device code to handle the asynchronous height updates. The main issue with this approach is that it requires global atomic operations issued by the CPU, however the global atomic function in CUDA is only available in device code. Using compiler builtins for atomics also failed.

The second attempt tried to use blocking to implement global relabeling. After every $|V|/2$ iteration we would block all executing GPUs when the current running kernel is finished. Then we would run the global relabeling operation. Blocking the GPUs requires implementing a custom worker in Groute, and some problems occured during the end of executing preventing the program from terminating. The reason this attempt failed is less clear than the first attempt, and a part of the problem may be internal to Groute.

A part of the problem with using the distributed worklist provided by Groute is that it may end the computations if it blocks for 5 seconds. The error message provided if this happens is unclear and only suggests changing the sizes of the lists used for communication. Further attempts to solve this problem was not successful given the time limit set by NTNU for this thesis. However, we suggest a solution to this problem in the future work section in the final chapter.

We nevertheless managed to develop and implement a multi-GPU version of the above max flow algorithm which ran correctly on most of the benchmarks we tested. Results and analysis of these benchmarks are provided in the next chapter.

Chapter 4

Benchmarking

In this chapter we describe and discuss the benchmarking results for our implementation on an NVIDIA DGX-2 on two, three, and four GPUs. The benchmarking is done using synthetic graphs, and the results are compared with a simple single thread CPU implementation and the single GPU implementation found in Gunrock. We first present the datasets, hardware, and software used during benchmarking. Then we present the results of the benchmarking, and discuss the results.

4.1 Datasets

We used three genrmf graphs of different sizes to see how well the implementation performs. Genrmf graphs are graphs made of square grids of vertices [3]. A single square grid of nodes is called a frame. Each node in a frame is connected to its neighbours and to a random node in the next frame. The source node is in a corner of the first frame, and the sink is in the opposite corner of the last frame.

Table 4.1: The graphs used during benchmarking.

Name	$\|V\|$	$\|E\|$	Frame size	Amount of frames
Small	1000	5400	10x10	10
Medium	2000	11000	10x10	20
Large	8000	45600	20x20	20

4.2 Hardware and Software

The implementation was benchmarked using an NVIDIA DGX-2 machine. The DGX-2 has 16 NVIDIA Tesla V100 GPUs [19] connected in an all-to-all topology using NVLink and 16 NVSwitches. A custom Docker container was built, and all benchmark were run inside this container. We also ran the benchmarks on Gunrock which is a single GPU graph processing framework[6], and a single thread version of the algorithm running on the CPU that was written to get experience with the algorithm before implementing it in Groute.

All relevant software versions are listed in table 4.2.

Table 4.2: The software versions used during testing. Note that the first two are git commit hashes.

Software	Version
Max Flow	f7a98f
Gunrock	138e85
Docker	19.03.4
GNU Make	4.1
G++	7.4.0
CUDA driver	418.116.00
Ubuntu	18.04 LTS
CUDA	10.1
CMake	3.10.2

4.3 Results

The results of the benchmark can be found in Fig 4.1, 4.2, and 4.3, and a comparison between our implementation, Gunrock's single GPU version which has global relabeling, and our simple CPU based version is found in table 4.3.

From this we can see that our implementation has extremely poor performance. On the largest graph the multi GPU version is over 17 times slower than the single thread CPU version. It should be noted that Gunrock does well in this comparison because it uses global relabeling, but it is beat by a single thread CPU implementation that also uses global relabeling.

There are many reasons why our implementation is so slow. The most obvious one is that global relabeling is not implemented. Another reason is the large amount of kernel launches needed to compute the max-flow. The algorithm executes at most $O(|V|^2|E|)$ operations [3]. Some of these are done in parallel, but even if every node is always active the lower bound on the amount of kernel launches is $\Omega(|V||E|)$. The irregularity of the algorithm also makes any attempt the GPU makes at memory coalescing pointless.

From the figure we can see that the current implementation does not scale. The reason it does not can be seen when looking at GPU utilisation and which nodes are active during execution. When looking at GPU utilisation we see that the GPU that has been assigned the nodes close to the sink is doing a lot of work while the other GPUs are barely doing any. This is supported by looking at the active nodes during execution. It seems that flow is quickly pushed to the sink, but pushing excess flow back to the source is a slow process.

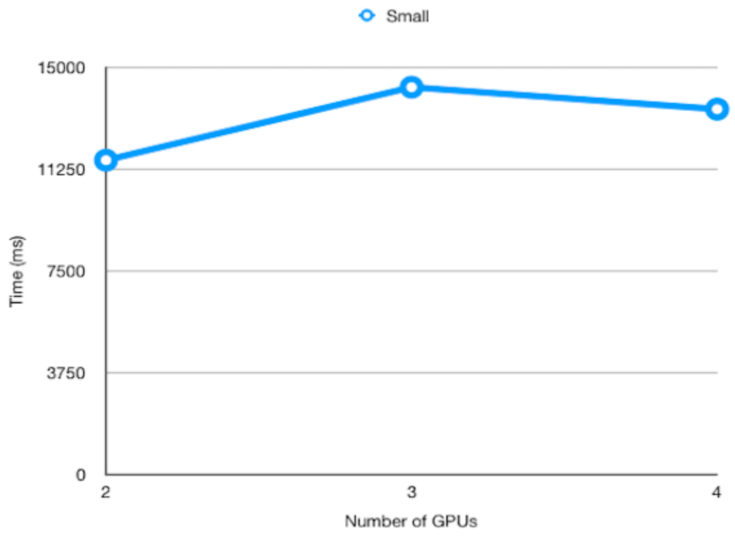

Figure 4.1: Runtime on the small dataset

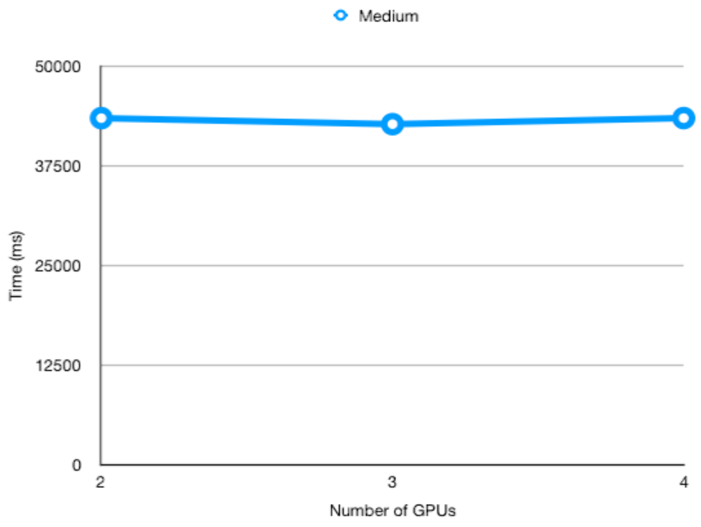

Figure 4.2: Runtime on the medium dataset

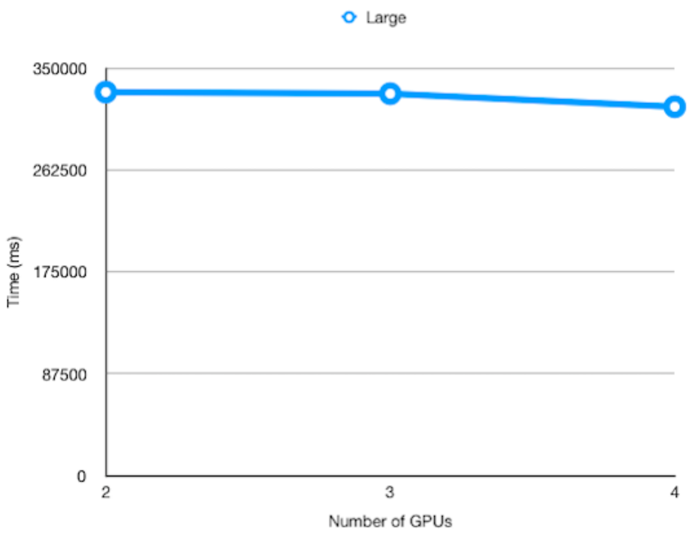

Figure 4.3: Runtime on the large dataset

There were some errors in the results encountered during execution. This only happened when benchmarking using the medium graph on two and four GPUs. The error rate during the initial testing of eight runs were $\frac{4}{8}$ on two GPUs and $\frac{3}{8}$ on four GPUs. A second error testing run of 20 runs resulted in 6 errors on two GPUs and 14 errors on four GPUs.

Table 4.3: Comparison of runtime between different max flow implementations

Implementation	Small	Medium	Large
Multi-GPU	11568 ms	42683 ms	317229 ms
Gunrock	43.13 ms	52.38 ms	522 ms
CPU	429 ms	2567 ms	18128 ms

4.4 Discussion

Our initial goal was to find the suitability of a preflow-push algorithm for multiple GPUs. From the results we can gather that it is possible to implement this algorithm and get correct results, but more efforts need to be put into both optimization and ensuring program correctness. We can't tell whether a more optimized multiple GPU implementation may be able to beat multithreaded CPU implementations.

A reasonable argument can be made that we should have benchmarked the implementation on more graphs. This is a very valid opinion, but we believe that the presented benchmarks accurately show the performance and the problems of our implementation accurately.

If we were to try implementing a max flow algorithm on multiple GPUs again we would first either start off with a more updated framework, or just rely on the CUDA managed memory for communication. The most difficult part of programming multiple GPUs is getting the data flow and communication right, and handing this problem over to the driver engineers may be a better choice than doing it ourselves. We would also focus on getting the correctness of the implementation down before moving on to attempting adding global relabeling. A lot of time was spent attempting to add this while it could have been spent on getting a better understanding of Groute and its codebase. This would help while chasing down and fixing any bugs that shows up.

Chapter 5

Conclusion and Future Work

Maximum flow is a graph algorithm with applications in many graph flows, graph cuts, scheduling, optimization and more [2]. We present what to our knowledge is the first multi-GPU max-flow implementation. Multi-GPU systems have become quite common due to the large amount of compute power they offer. This amount of power is offset by the difficulty of programming such systems.

We implemented the asynchronous nonblocking multithreaded algorithm by Hong and He using Groute. This is a preflow-push algorithm that can asynchronously push flow and relabel nodes. We attempted to add the global relabeling heuristic to our implementation, but failed on two occasions. The first failed attempt tried to do global relabeling asynchronously on the CPU. This failed because it requires the CPU to issue system wide atomic instructions which is only available in CUDA device code. The second attempt would block all kernel execution after a set amount of iterations. The problem behind this failure is less clear, and may be an internal Groute problem.

A part of the problem with using the distributed worklist provided by Groute was that it may end the computations if it blocks for 5 seconds. The error message provided if this happens is unclear and only suggests changing the sizes of the lists used for communication. We nevertheless managed to develop and implement a multi-GPU version of the above max flow algorithm which ran correctly on most of the benchmarks we tested.

We benchmarked the implementation on an NVIDIA DGX-2. The result of this benchmarking is that the implementation is not error proof, and has poor performance compared to a simple CPU implementation. The implementation also does not scale when more GPUs are added which is a result of the simple graph partitioning algorithm used. Our disappointing results show how challenging it is to implement irregular algorithms efficiently on multiple GPUs.

Finally we propose a third possible way to implement global relabeling by using the GPU instead of the CPU, and a way to help the load balancing issue by changing the partitioning algorithm.

5.1 Future Work

There are many ways one can try to optimize and fix the current implementation. Fixing the bugs would probably require updating Groute to use newer CUDA features to replace the deprecated ones. Doing this requires a deeper knowledge of the inner workings of the framework and its dependencies.

The first step to optimize the implementation would be to add proper global relabeling. A different approach to this that may have more success than the two approaches already discussed is to do it on the GPU. By doing global relabeling on the GPU you now have access to the global device atomics in CUDA that are not available on the CPU. There is also less communication between the CPU and the GPU since the updated distance labels are already on the GPU. The communication speed between GPUs may also be higher if they are connected with NVLink compared to being connected over PCIe.

Another way to reduce the running time would be to change the partitioning scheme. The current sequential scheme provides poor load balancing and does not try to minimize the amount of edges cut. With fewer edges cut there are less chances for multiple GPUs to simultaneously push flow over an edge reducing synchronization costs and error rates. Another way this may reduce the running time is that if the GPU knows that it is the only owner of the edge flow is being pushed over, the GPU may use local atomic operations instead of global ones. This cuts any potential GPU to GPU communication a global atomic operation may need.

Bibliography

[1] Nvidia Corporation. NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. [Online; accessed 16-October-2019].

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Trans. Parallel Distrib. Syst.*, 22:1025–1033, 06 2011.

[4] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 235–248, New York, NY, USA, 2017. ACM.

[5] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[6] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yudua Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.*, 4(2), September 2017.

[7] Vegard Seim Karstang and Anne Cathrine Elster. Evaluation of GPU-based graph frameworks, 2019.

[8] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[9] Nvidia Corporation. NVIDIA Tesla V100 GPU Accelerator. `https://images.nvidia.com/content/technologies/volta/pdf/`

437317-Volta-V100-DS-NV-US-WEB.pdf, 2017. [Online; accessed 16-July-2019].

[10] Nvidia Corporation. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. [Online; accessed 16-October-2019].

[11] Nvidia Corporation. NVIDIA NVSwitch. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018. [Online; accessed 17-October-2019].

[12] Nvidia Corporation. About CUDA. https://developer.nvidia.com/about-cuda. [Online; accessed 16-July-2019].

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[14] Albert-László Barabási. Scale-free networks: A decade and beyond. *Science*, 325(5939):412–413, 2009.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[16] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[17] Yossi Shiloach and Uzi Vishkin. An o(n2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128 – 146, 1982.

[18] Richard J. Anderson and João C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, page 168–177, New York, NY, USA, 1992. Association for Computing Machinery.

[19] Nvidia Corporation. NVIDIA DGX-2 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf, 2019. [Online; accessed 04-July-2019].

Appendix A

CPU benchmarks

We implemented a multithreaded version of the algorithm before starting on the multi-GPU version to gain experience and knowledge about the algorithm. During this implementation process we ended up with several versions that were all benchmarked. The benchmarking of the multithreaded version was done on a 16 core Ryzen 7 1800x running at 3.6 GHz, with 16 GB of DDR4 2400 MHz RAM. The results of this benckmarking and a comparison with Gunrock and our multi-GPU version is found in Table A.1. From these results we can clearly see the effect global relabeling has on the runtime of the algorithm. The code can be found at github.com/vegaka/Graphs.

Version	Small	Medium	Large
Multi-GPU	11568 ms	42683 ms	317229 ms
Gunrock	43.13 ms	52.38 ms	522 ms
Single thread	429 ms	2567 ms	18128 ms
Single thread with global relabeling	19 ms	34 ms	275 ms
16 threads with global relabeling	16 ms	27 ms	288 ms

Table A.1: Benchmark results for various implementations

Appendix B

GPU kernel code

We list the code that processes a node in Listing B.1. The full code can be found at github.com/acelster/Groute-Astar-MaxFlow/ under the maxflow tree. Access can be requested from Dr. Elster. The code is run for every item supplied by the distributed worklist and produces new items. Note the similarity between the implementation and the pseudocode in Algorithm 2.

Listing B.1: The part of the kernel that processes a single node

```
if (excess[node] > 0) {
    int h = INT32_MAX;
    index_t next_node = UINT32_MAX;
    index_t lowEdgeId = UINT32_MAX;
    index_t begin_edge = graph.begin_edge(node);
    index_t end_edge = graph.end_edge(node);

    if (begin_edge == end_edge || node == s || node == t) {
        is_active[node] = CUDA_FALSE;
        continue;
    }

    for (index_t j = begin_edge; j < end_edge; ++j) {
        index_t n = graph.edge_dest(j);

        if (residuals[j] <= 0) continue;

        if (heights[n] < h) {
            next_node = n;
            h = heights[n];
            lowEdgeId = j;
        }
    }
    if (heights[node] > h) {
        int delta = min(excess[node], residuals[lowEdgeId]);
        atomicSub_system(&residuals[lowEdgeId], delta);
        atomicAdd_system(&residuals[reverse[lowEdgeId]], delta);
        atomicSub_system(&excess[node], delta);
        atomicAdd_system(&excess[next_node], delta);

        if (!atomicCAS_system(&is_active[next_node],
            CUDA_FALSE, CUDA_TRUE)) {
            work_target.append_work(next_node);
        }

        if (excess[node] > 0) {
             work_target.append_work(node);
        } else {
            is_active[node] = CUDA_FALSE;
        }
    } else {
        heights[node] = h + 1;
        work_target.append_work(node);
    }
}
```