**DENSEST SUBGRAPH DISCOVERY ON THE GPU**

by

Hunter G. Gareau

A Thesis

Submitted to the

Department of Computer Science

College of Science and Mathematics

In partial fulfillment of the requirement

For the degree of

Master of Science in Computer Science

at

Rowan University

(insert defense date)

Thesis Chair: Guimu Guo, Ph.D., Assistant Professor, Department of Computer Science

Committee Members:

Sihan Yu, Ph.D., Assistant Professor, Department of Computer Science

Richard Rabbitz, MSc., Assistant Teaching Professor, Department of Computer Science

© 2025 Hunter G. Gareau

**Acknowledgements**

I would like to deeply thank Professor Guimu Guo for his guidance throughout our research. We have worked together for a long time on this project, and I wouldn’t have been able to do any of this without his skills, direction, and help.

I would like to thank Professor Sihan Yu and Professor Richard Rabbitz for being a part of my committee, being available to assist me in this endeavor, and for reading over and reviewing my paper.

I would also like to thank Wajid Manzoor, a Ph.D. student and fellow student of Professor Guo, who was a massive help on developing the CUDA code.

# **Abstract**

Hunter Gareau

DENSEST SUBGRAPH DISCOVERY ON THE GPU

2024 – 2025

Guimu Guo, Ph.D.

Master of Science in Computer Science

The Densest Subgraph Discovery (DSD) problem is a prevalent problem in the field of graph mining, aiming to find the subgraph of highest edge to vertex ratio. There have been many solutions to this problem. However, one avenue that other graph mining problems have gone down is parallel programming, which executes many commands simultaneously for more efficient programs. What we have done is applied parallel programming on the GPU to a DSD solution called CoreExact in a two-fold effort. This is done in hopes of both creating a more efficient solution to the DSD while also serving as further research and precedent for the efficacy of parallel programming using the GPU.

**Table of Contents**

[Abstract iv](#_Toc198648409)

[List of Figures vi](#_Toc198648410)

Chapter 1: [Introduction 1](#_Toc198648411)

[The Densest Subgraph Discovery Problem 1](#_Toc198648412)

[The GPU And Parallel Programming With CUDA 3](#_Toc198648413)

Chapter 2: [Existing Solutions and Related Works 6](#_Toc198648414)

[Goldberg’s maximum flow-based algorithm 6](#_Toc198648415)

[Greedy Peeling Algorithm 7](#_Toc198648416)

[The Greedy++ Algorithm 8](#_Toc198648417)

[Densest k-Subgraph Approximation 9](#_Toc198648418)

[DSD for Directed Graphs 10](#_Toc198648419)

[Optimal Quasi-clique Problem 11](#_Toc198648420)

Chapter 3: [The CoreExact Algorithm 12](#_Toc198648421)

[Adjacency Lists and Adjacency Matrices 12](#_Toc198648422)

[K-Cores and Decomposition 14](#_Toc198648423)

[Connected Components 18](#_Toc198648424)

[Flow Networks and Min st-Cuts 19](#_Toc198648425)

[An in Depth Look at Goldberg’s Algorithm 23](#_Toc198648426)

[The Full Algorithm 26](#_Toc198648427)

Chapter 4: [Our Parallelized Implementation 28](#_Toc198648428)

[DAG Generation 29](#_Toc198648429)

[Finding all Cliques 31](#_Toc198648430)

[Clique-based Core Decomposition 32](#_Toc198648431)

[Construction and Pruning of the Densest k-Core 34](#_Toc198648432)

[Component Decomposition 37](#_Toc198648433)

[Bound Generation 39](#_Toc198648434)

[Flow Network Creation 40](#_Toc198648435)

[In Summary 41](#_Toc198648436)

Chapter 5: [Experiment and Evaluation 42](#_Toc198648437)

[References 42](#_Toc198648438)

Please do not italicize any of the entries in your Table of Contents.

Please include chapter numbers and titles in your Table of Contents. All chapter headings and titles should remain on the same line, separated by a colon, in the Table of Contents. (Example: Chapter 1: Introduction; Chapter 2: Literature Review; etc.) (If you are using the Word’s automatic ToC generator, you can use the work-around outlined in red in this guide to format your chapter headings correctly in both the text and the ToC.)

Please capitalize the first letter of each important word in your headings, both on the ToC and in the headings as they appear in the text.

# **List of Figures**

[Figure 1. DSD Example. 2](#_Toc198215988)

[Figure 2. GPU Architecture [5]. 5](#_Toc198215989)

[Figure 3. Adjacency List Representation [9]. 13](#_Toc198215990)

[Figure 4. Adjacency Matrix Representation. 14](#_Toc198215991)

[Figure 5. Graph with outline of 1-core, 2-core, and 3-core [11]. 15](#_Toc198215992)

[Figure 6. Connected components example [12]. 18](#_Toc198215993)

[Figure 7. Flow Network example [13]. 20](#_Toc198215994)

[Figure 8. st-cut example. 21](#_Toc198215995)

[Figure 9. Minimum st-cut example. 22](#_Toc198215996)

[Figure 10. CoreExact flow network. 24](#_Toc198215997)

[Figure 11. CoreExact algorithm [2]. 27](#_Toc198215998)

[Figure 12. k-clique examples [17]. 29](#_Toc198215999)

[Figure 13. Processing vertex neighbors by thread. 30](#_Toc198216000)

[Figure 14. Processing vertices by thread. 33](#_Toc198216001)

[Figure 15. Handling cliques by warp and its edges by thread. 36](#_Toc198216002)

[Figure 16. Handling components by warp. 40](#_Toc198216003)

[Figure 17. Handling components by warp and their vertices by thread. 41](#_Toc198216004)

(update)

**List of Tables**

Table Page

Table 1. (Title here) (Page #)

**Chapter 1**

# **Introduction**

When it comes to analyzing large or complex groups of data, it is often useful to examine the connections and relationships shared between its subjects. And graphs can be used to model such relations. Graphs are composed of two components: vertices (which represent individual members of a data set) and edges (which represent the connections between these members). A graph is usually represented as *G = (V, E)*, where *V* is the set of vertices in the graph, and *E* is the set of edges in the graph. A couple of the major types of data that graphs can be used to model are social networks (e.g. Facebook, Twitter, etc.) and biological data (DNA, neural networks, etc.) [1]. It’s also worth noting that edges can be directed, as in a connection between two vertices specifically goes from one to another. A directed graph can be used to model other types of data sets, or specific types of relations, such as people following others in an online social network. The analyzation of these graphs using various tools or techniques to find additional data and patterns is known as Graph Mining.

## **The Densest Subgraph Discovery Problem**

While there are many studies and problems in graph mining, a fundamental one is known as the *densest subgraph discovery problem* (the DSD). The aim of the DSD is that given an undirected graph *G*, you must find a subgraph *S* such that it has the highest density of all subgraphs of *G*. The definition of edge density is as follows:

Definition 1 (Edge Density [2]): Given a graph G (V, E), edge density is

Additionally, density can also be applied to network motifs, which are small structures of vertices and edges such as shapes or cliques:

Definition 2 (Motif Density [2]): Given a graph G (V, E) and its set of motifs of a chosen type M, its motif density is .

Motif Density can be applied as edge density or clique density (when your motif type is “edge”), so it is the definition we will use. So, we can define the DSD as such:

Definition 3 (Densest Subgraph Discovery Problem): Given a graph G (V, E) and its set of motifs of a chosen type, the densest subgraph *S* is such that and the motif density of all other subgraphs of G.

A diagram of a network

AI-generated content may be incorrect.

Figure 1. DSD Example.

The denser a graph is, the more connected the members of that graph are. So, in simple terms, the DSD aims to find the most connected group of vertices within a graph. An example of a densest subgraph is found in Figure 1, where the subgraph of the pink vertices is denser than all other subgraphs in the graph. The densest subgraph (and therefore a solution to the DSD) is a notable piece of information to have for a dataset and has plenty of notable applications in real data sets including finding and filtering out fake users or identifying echo chambers in social networks, or identification of regulatory motifs in DNA or gene annotation graphs in biological data [1].

## **The GPU And Parallel Programming With CUDA**

Being such a notable problem, there are of course many solutions to the DSD. However most, if not all, are serialized programs that run on the CPU. By the nature of graph mining, it is almost always required to process every vertex and edge in these large graph datasets, which can certainly take time in a serialized program processing these one by one. But by programming in parallel, a great number of vertices can be processed in concurrence and thus save time and be more efficient. This is where the GPU comes in. The CPU and GPU differ in their processing cores. The CPU runs on a handful of powerful processing cores that can take complex orders, while the GPU runs on many weaker processing cores that take simpler orders. So, while a powerful CPU can run a taxing program efficiently, utilizing the full power of a GPU by running commands in parallel across its many cores is much more computationally efficient. Although this comes with the limitation that the GPU’s commands are much more restrictive than that of the CPU [3].

These processes and commands are all run across threads, which can be thought of as a “unit” of execution within an overall process with its own data register. For example, if you had an array of integers that you needed to add 10 to individually, each thread of the GPU would take an integer from the array and add 10 to it at the same time. These threads are grouped hierarchically. 32 threads make up a warp, with every thread in a warp following the same instructions. Data is easily shared between thread registers in a warp. A step up from this, some number of warps make up a block (also referred to as a thread block). The number of warps per block varies between GPUs, and the blocks are mapped to the processors of the GPU. There is a cache of shared memory for the warps and threads in a block, but the collective register memory of every thread is often greater than the amount of available memory in the block’s cache. So, data can be shared between warps in a block, but it is more computationally taxing. The final level of the hierarchy is grids, which are made up of blocks. When a kernel (a process to be run in parallel) is launched that kernel gets mapped to a grid, using the processors mapped to the grids blocks to complete the task. The entire GPU shares an L2 cache and a global memory, but sharing memory across blocks, warps, and threads using this memory is taxing due to the synchronous nature of the threads and memory updates being dependent. This architecture is visualized in Figure 2. With this architecture, work and different processes can be divided up between blocks, warps, and threads, allowing the user notable control and options over what can be processed and how it can be worked with in parallel [4].

A diagram of a computer code

AI-generated content may be incorrect.

Figure 2. GPU Architecture [5].

So, the computational power and structure of parallel programming on the GPU is well suited for graph mining, and thus the DSD. With the complexities of graph mining and the DSD however, one would have a very tough time writing a program to analyze a graph only using the simple commands available to the GPU. But thanks to NVIDIA, there’s a tool to work around that. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface model. CUDA allows for a serialized C++ program run on the CPU to execute threads in parallel on the GPU and transfer data between the CPU and GPU, allowing the user to leverage the massive computational power of the GPU as needed [3]. This interaction between serialized code and running kernels on the GPU can be seen in Figure 2.

And that brings us to the goal of the project. Programming a solution to the DSD which utilizes the GPU through CUDA. This goal is two-fold, both being a more efficient solution to the DSD, as well as providing further research on the computational power of parallel programming on the GPU.

**Chapter 2**

# **Existing Solutions and Related Works**

As stated, there are many existing solutions to the DSD. Algorithms that have been created and developed to be more and more efficient. Some algorithms will get the exact solution to the DSD, meaning these algorithms will return the densest subgraph without fail. Other algorithms will approximate the densest subgraph, finding one of the densest subgraphs, if not the densest. So, it’s worth covering some of these algorithms to see how others have solved it before, and to introduce some important topics and ideas.

### **Goldberg’s maximum flow-based algorithm**

This algorithm was first developed by Andrew Goldberg in 1984. The basic outline of this algorithm follows the construction of a flow network based on the given graph, where every vertex is connected to two newly added vertices *s* and *t* (the source and sink). A binary search is run on this network, maintaining an upper and lower bound on the greatest density, and tightening these bounds with every iteration until the lower bound is within the margin of being exact. The flow network and upper and lower bounds are updated using a maximum flow (or min st-cut) approach which will try to return a subgraph of density equal to the average of the upper and lower bounds [4]. The time complexity of this algorithm depends on the implementation of the min st-cut, to the point where the complexity given to Algorithm 1 (Goldberg’s algorithm) of [4] is such that the time complexity is given by where T is the complexity of the min st-cut algorithm used and n is the number of vertices, since the min st-cut depends on the algorithm, and it’s log(n) iterations of the min st-cut algorithm since it’s a binary search on the size of the graph. There is a lot to unpack and understand with this algorithm, but these topics will be explored more in depth in the next chapter.

### **Greedy Peeling Algorithm**

This is a straightforward approximation algorithm developed by Moses Charikar in 2000. Given graph *G*, the algorithm will remove the vertex of lowest degree from *G* every iteration, where degree is the number of vertices a given vertex is connected to. The current subgraph of highest density is stored, and every iteration it is compared to the new subgraph. If the new subgraph is denser than the currently stored one, it replaces it. This goes on until all vertices have been removed, and thus the subgraph that had the highest density as these vertices were peeled away is the result. As is apparent, this is not a very reliable algorithm to find the densest subgraph, as it leaves a lot of possible subgraphs unchecked. However, it was found and proven that through this method, the resulting subgraph would always be *at least* half as dense as the actual densest subgraph [4]. Meaning this is a decent algorithm for finding dense subgraphs, and there is notable idea in the idea of peeling off vertices by lowest degree. Additionally, its time complexity is straightforward, being where n and m are the number of vertices and edges for a non-weighted graph. This complexity is due to the fact that after the initial scan for degrees, each vertex and edge is only operated on once, being when they are removed, thus the number of steps is reliant on the number of vertices plus the number of edges [4].

### **The Greedy++ Algorithm**

Building on Charikar’s Greedy Peeling Algorithm is an algorithm proposed by D. Boob, Y. Gao, R. Peng, and J. Wang in 2019. This algorithm iterates through the Greedy algorithm *T* times, with each iteration updating the priority of each vertex so that in subsequent iterations, vertices of higher priority are kept in for longer. By running the Greedy algorithm multiple times and utilizing the results of previous iterations, denser subgraphs can be found. The time complexity for this is similar to the original Greedy algorithm with , but with two notable changes. First, since priority is being used, each selection of the lowest degree vertex takes a bit of additional time to determine and update priority, coming out to time in a binary search for each vertex, getting us to . And since we are doing a chosen number *T* iterations, we find that the time complexity of Greedy++ is . Additionally, while this algorithm is still an approximation, it was found that with enough iterations, the densest subgraph found would be arbitrarily close to the exact optimal densest subgraph. Meaning this relatively simple solution can find a subgraph of negligible difference from the optimal densest subgraph at worst, making for an efficient and relatively easy to implement solution to the DSD [4].

### **Densest k-Subgraph Approximation**

This is a notable variant of the DSD where the subgraphs being searched for are specifically of size *k* (subgraphs with *k* vertices). This can be useful information in various contexts and is also an interesting problem to dissect on its own since it essentially reduces the DSD from looking for to just In other words, you’re looking for the subgraph of size *k* with the most edges.

There are several existing solutions to this problem, but one of the major ones was developed by U. Feige, G. Kortsarz, and D. Peleg in a combinatorial approximation algorithm [5]. The basic algorithm (referred to as Algorithm A in the paper) combines three procedures for finding a dense subgraph of size *k* and returns the densest of the three. The first procedure simply takes *k/2* random edges returns the set of vertices connected by these edges, adding additional arbitrary vertices to get the set to size *k* if needed. This procedure provides a baseline, always returning a subgraph of density ≥ 1. The following 2 procedures act as Greedy approximations to try and find an even denser subgraph. The first of the two sorts the vertices by degree, and takes the k/2 vertices of highest degree into the subset, then resorts the remaining vertices by how many neighbors they have in the initial subset, and adding the last k/2 vertices from the top of that ranking. The second of these procedures constructs a subgraph for every vertex *v* in the graph, being constructed by ranking how many 2-step paths each vertex has to *v* and then ranking the neighbors of *v* by how many of those vertices they’re connected to, followed by taking the k/2 vertices of the highest degree connected to those vertices. The union of this set and the 2-step neighbors serves as the resulting subgraph, adding in arbitrary vertices if the result does not reach k. As stated before, the result of these three procedures with the greatest density is taken as the result of the algorithm. This algorithm is of accuracy . In the same paper, they dive into ways to approximate even closer, and other algorithms have been made to approximate the densest k-subgraph which have a higher accuracy, but this algorithm served as the first notable solution.

### **DSD for Directed Graphs**

As was mentioned earlier, the definition of density we utilize only applies to undirected graphs, as that definition does not take directionality into account. A definition for density in directed graphs was proposed by Kannan and Vinay in 1999 [6], and is still in use today. It is defined as follows:

Definition 4 (Directed Graph Density [6]): Given a directed graph G (V, E), let and E (S, T) is all the edges from S to T. We define the density of these sets as

So in layman’s terms, directed graph density represents how connected one set of vertices is to another. And as follows, the Directed Densest Subgraph Problem (DDSP) aims to find sets such that *d(S, T)* is maximized.

There are multiple solutions to this problem as one may expect, and one such algorithm developed by Charikar will find the exact solution. This solution is a Relaxed Linear Programming problem based on the value of *|S| / |T|* (referred to in the paper as *c*). As proven in [7], the optimal value of the linear programming problem on *c* is equivalent to the optimal directed density, where the optimal sets *S* and *T* can be computed from the results of the LP.

### **Optimal Quasi-clique Problem**

The definition of density most widely used is the one covered in Definition 1. However, some have put forward different definitions to gather different results. One such definition is called Edge-Surplus:

Definition 5 (Edge – Surplus [8]): Given graph G (V, E), let , where S (X, Y). Given some functions g and h, and some > 0, we define edge surplus as:

. In the case of .

This definition is flexible, as the definition of *g*, *h*, and are determined for the specific case. But this framework sets up the problem such that it favors more edges and penalizes more vertices (hence why is positive, and why is negative).

This definition can be used to evaluate the DSD as normal, but in the case of quasi-clique: , , and These definitions favor subgraphs that are tighter knit and have a small diameter (subgraphs with longest paths that are smaller). This differs from the normal density definition which does not distinguish graph size. And as follows, the Optimal Quasi-clique Problem (OQP) aims to find the subgraph that maximizes the value of this function. One such solution to OQP is based on the Greedy peeling algorithm for the DSD. The main differences lie in that obviously this algorithm checks for quasi-clique value rather than standard edge density, and that they increase efficiency by keeping lists of all possible degree values, updating them as vertices are removed and using them to decide which vertex to remove next [8].

**Chapter 3**

# **The CoreExact Algorithm**

While these various solutions are all very interesting algorithms and problems to discuss, let’s move on to the most important algorithm of our project. Our program is based on the CoreExact algorithm [2]. We will be parallelizing the algorithm in a C++/CUDA program, but to understand the implementation, we should first explain CoreExact. There are a lot of concepts and smaller algorithms that make up CoreExact though, so we will go into each of these before putting it all together.

## **Adjacency Lists and Adjacency Matrices**

Because algorithms must be applied to actual code, there needs to be implementations of graphs programming-wise. There are multiple ways to do so, but the method used for graphs in our program (and in the CoreExact program) is the Adjacency List. This is a straightforward way of storing how vertices are connected using a 2-dimensional list (a list of lists). The first dimensional list represents every vertex in the graph, where the index matches the vertex number. The second dimension is a list of all the vertices that the represented vertex is connected to. A representation of this is shown in Figure 3. Of note, because our graph is undirected, the edge will be stored bidirectionally, so if an edge connects vertices 1 and 2, 2 and 1 show up in their lists respectively.



Figure 3. Adjacency List Representation [9].

Additionally, since CoreExact is allowing for the use of motifs other than regular edges, the motif structures also need to be stored. In theory, adjacency lists could be used here as well. However, CoreExact uses adjacency matrices to do so. Adjacency matrices, in comparison to adjacency lists, work well for small structures like motifs as they make construction and representation easier in the code (which holds true for large graphs as well, but they become much more costly in memory and iteration efficiency).

As the name suggests, the adjacency matrix represents graphs through a full 2-dimensional matrix. There is a row and column for every vertex, with each intersection being filled with a 0 or 1. If the intersection has a 0, the two vertices are not connected by an edge, whereas a 1 indicates they are connected. A representation of this is shown in Figure 4, where the graph on the left has its edges represented by 1’s in the table matrix.

A diagram of a network

AI-generated content may be incorrect.

Figure 4. Adjacency Matrix Representation.

In the actual code, both adjacency lists and matrices are handled as 2-dimensional vectors, but their structures still reflect what has been outlined above.

## **K-Cores and Decomposition**

A major aspect of the CoreExact methodology is that we can prune down the input graph to a subgraph, and break that down to save time searching. The first of these steps is finding the densest k-core.

A k-core is a graph in which every vertex is connected to at least *k* other vertices. So in a 1-core, every vertex is connected to at least one other vertex. In a 2-core, every vertex is connected to at least 2 other vertices. And so on. But within a large graph, there are likely to be subgraphs of a much higher k value. To find these graphs, we can use a peeling technique like the Greedy peeling algorithm covered earlier called core decomposition. Like with the Greedy algorithm, core decomposition finds the vertex of lowest degree one at a time and removes it from the graph, where degree is the number of vertices a vertex is connected to. Using a 1-core as an example, if you remove all the vertices of degree 1 (and all subsequent vertices that have degree 1 after the removal of the other vertices), you will be left with only vertices that have degree 2 or higher, going from a 1-core to a 2-core. Going until all vertices have been removed will mean every k-core up to the highest possible *k* value will have been found. So, using this method you can record the information on all these k-core subgraphs, and as stated earlier, this can be done in time (going forward this is how we will refer to the number of edges *e* and number of vertices *v*).

Figure 5 outlines an example of a graph which can be broken down into a 1-core, 2-core, and 3-core. The orange vertices are in the 1-core since they all have a degree of 1 (they’re only connected to 1 vertex) and are removed at the first stage. The blue vertices are in the 2-core since they only have a degree of 2 after the orange vertices’ removal. This leaves the red vertices, which are all connected to 3 vertices each, leaving a 3-core.



Figure 5. Graph with outline of 1-core, 2-core, and 3-core [11].

And this process and terminology applies to any motif, not just edges. So a 1-core of triangles means every vertex is part of at least one triangle, a 2-core means every vertex is part of at least 2 triangles, and so on. The decomposition process is the same as well, since you remove vertices one at a time based on motif degree. And motif core decomposition is the first step of the CoreExact algorithm.

However, to begin motif core decomposition in the code, we need to find every motif in the graph (since individual motif data is not stored for each graph, only edges). To do so, you first need to generate a DAG of the graph (directed acyclic graph). Finding motifs requires branching down all possible paths, meaning any loops in the graph would cause potential repeats. And DAGs are designed such that they have no loops. In our specific case, we set it up so that all edges are directed from the vertex of higher degree to the vertex of lower degree. This means our first step is regular edge decomposition, which we’ve already covered. Then it’s a simple process of generating a “new” graph where each edge is only stored from the higher degree vertex to the one of lower degree, rather than being stored for both directions as we covered in our adjacency lists section.

Once the DAG is created, we can find all the motifs in the graph. This process is done with a recursive algorithm. It is called on the DAG, and we pass the size of our chosen motif to it, *k*. By knowing *k,* we can branch down paths in the DAG that only result in a list of vertices of size *k*, effectively making it so we only check possible motifs. This algorithm can be divided into two cases: if *k* is equal to 2 (our motif is edges), or not. In the former case, for every neighbor of the current vertex, we have found a complete motif. So for each neighbor, we update the number of motifs found, the motif degree of the vertex, and we add the path of vertices to our motif list. In the latter case, we check every vertex in our current cluster of neighbors (this starts as the list of all vertices in the DAG). For each vertex, we add all neighbors of the vertex to a new list that are valid for the current recursive level *k* (level of connection to other vertices). Then after updating the degrees of every neighbor to reflect the next recursive level, we make our recursive call, passing our current path, *k* – 1 (to reflect the next recursive level), and our list of valid neighbors. This is an in-depth algorithm, but all that’s important to know is that it finds all the motifs in the graph, and stores motif information such as the number of motifs in the graph and the motif degrees of every vertex.

Once your motifs information is gathered, it’s just a matter of doing motif core decomposition, which as we covered, is basically the same as edge decomposition. So in order, we do edge decomposition, DAG generation, motif listing, and then motif core decomposition can be performed, resulting in finding every motif k-core. And the densest of these k-cores must contain the densest subgraph, as proven in [2]. This comes down to two steps. First, proving the lemma that removing any vertex from the densest subgraph will result in several motifs being removed equal to the ceiling value of the density of the densest subgraph (we’ll call this value ⌈*p*⌉). This was proven by contradiction. If you assume that the statement is false, and that less than ⌈*p*⌉ motifs are removed, you get a subgraph with a higher density than the previous subgraph. However, that is a contradiction, since we know that the previous subgraph was the densest subgraph. Proving the lemma. So intuitively, this means that any given vertex in the densest subgraph is a part of ⌈*p*⌉ motifs. Which is the same as the definition of a k-core, where any vertex has participated in at least *k* motifs. Thus, we can conclude that the densest subgraph must be in the k-core where k = ⌈*p*⌉.

## **Connected Components**

With the densest k-core, we can break this down into pieces known as connected components. A connected component is a subgraph where every vertex is connected to all the others in the subgraph by *some* path. In Figure 6 is an example of a graph of connected components, where the graph can be broken into 3 disjointed subgraphs *{V1, V2, V3, V4, V5, V6}, {V7, V8, V9}, and {V10, V11, V12}*. In layman’s terms, a connected component is a subgraph that has no connections to the vertices of the other connected components. Not every graph is going to have connected components, but it’s worth checking for, as the connected components of the densest subgraph have the same density as each other, and by breaking down the k-core, we may find certain connected components are denser than others [2].



Figure 6. Connected components example [12].

To decompose into connected components in the code, multiple steps occur. First off, connected components are found using a breadth first search, which checks each vertex and edge once, resulting in time complexity. However, after this the connected component subgraphs need to be constructed and added to a list. Most of this is negligible for time complexity, save for working with the motifs. Processing motifs requires processing each vertex of each motif. So, for the number of motifs *m* and the size of our motif *s* we have complexity . Since the rest of the process is negligible, our time complexity comes down to .

## **Flow Networks and Min st-Cuts**

One of the most important pieces of CoreExact is the usage of Goldberg’s maximum flow algorithm. A key aspect of that algorithm is a constantly updating flow network using the next best guess for greatest density, and then using min-st cut to try and return a subgraph of at least that density. So, to understand how Goldberg’s algorithm works, it’s important to understand what flow networks are, and what a minimum st-cut is.

A flow network is a special type of directed graph where every edge has two values: capacity and flow. These are abstract numerical values that represent an amount of “something” passing through the edge in the direction it faces. Water through a pipe is an easy way to look at this, since water flows through a pipe in a certain direction. Capacity represents the maximum amount of this “something” that can pass through the edge, while flow is an amount greater than or equal to 0 that is less than or equal to the capacity, essentially representing how much of this “something” is passing through the edge. This is often represented as *x/y*, where *x* is the flow and *y* is the capacity. Using the water pipe example, a pipe may be able to sustain up to 5 liters of water at any moment, but the amount of water passing through may be anywhere from 0 to 5 liters. Flow networks have two special vertices called the source and the sink, usually referred to as *s* and *t* respectively. The source is where all flow starts and comes from, and the sink is where all the flow ends up. Another important detail is that flow follows the rules of conservation, where the amount of flow going into any given vertex must be equal to the amount coming out of it (unless they are the source or sink).



Figure 7. Flow Network example [13].

Shown above in Figure 7 is an example of a flow network. As can be seen, every edge has a flow/capacity pair, with the flow ranging from 0 to the capacity of that edge. We can see that flow only comes out of source s, and flows into sink t. Additionally, flow conservation can be examined here. As an example, Vertex A receives 8 flow (5 from s and 3 from D), and outputs 8 flow (5 to B and 3 to C). This rule is held up in the other vertices B, C, and D as well.

An st-cut is the division of a flow network into 2 subgraphs *S* and *T*, where *S* contains the source, and *T* contains the sink. This cut is made by removing edges such that there is no remaining connection between the two subgraphs (disjointing them). Cut capacity is the sum of capacities of the removed edges, but an important note is that only the capacity of edges that flow into *T* are counted in cut capacity. This can be examined in Figure 8. A cut of the network is being depicted such that the edges connecting 0 to 2, 2 to 1, and 1 to 3 are removed from the graph. This leaves two subgraphs, *S = {s, 0, 1}* and *T = {2, 3, t}*. We can calculate the cut capacity to be 5, since we ignore the capacity of edge 2 to 1 as it flows into *S*, leaving capacities of 2 and 3.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 8. st-cut example.

The minimum st-cut is the st-cut where cut capacity is minimized. Importantly, it’s been found that the value of this cut is equal to the maximum flow of the network [14], and so the problems can be used interchangeably. Using the same graph, we can find that the minimum st-cut is pictured in Figure 9, where we are left with *S = {s, 0}* and *T = {1, 2, 3, t}.* The cut capacity comes out to be 4, as the edge from 1 to 0 is excluded due to flowing into *S*, so leaving capacities of 2 and 2. If you examine the example graph further, you will find no st-cut with a cut capacity less than 4, which is why Figure 9 depicts the minimum st-cut.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9. Minimum st-cut example.

In the CoreExact algorithm, they use the Edmonds-Karp algorithm to find max flow, and therefore the min st-cut. Edmonds-Karp uses a breadth first search to find the shortest path from *s* to *t*, pass the max flow it can along this path, update the flow network to include this flow, then repeat until no more paths can be found. This results in a network of maximum flow, and by taking the st-cut of this graph, *S* (which will contain all vertices with flow running through them), will be the minimum st-cut subgraph. The time complexity of the Edmonds-Karp algorithm is [15]. This is because breadth first searches have time complexity of , but in this case, a large number of edges is the worst case, so we have worst case of for the breadth first search. But this has to run for every path, which in the worst case, is roughly equal to *ve* paths. So the worst case time complexity comes out to be .

## **An in Depth Look at Goldberg’s Algorithm**

Now that we’ve explained the necessary pieces, we can do an in-depth explanation of how Goldberg’s maximum flow algorithm works in CoreExact.

To start off, lower and upper bounds *l* and *u* are declared, and α is set as the average of *l* and *u.* Then, a specifically designed flow network that is based on any input graph is utilized. In this network, every edge (*u,v*) is replaced by a pair of directed edges, one from *u* to *v* and one from *v* to *u*, each with capacity 1. Two additional vertices are added to the network to be the source and sink (*s* and *t*). There is an edge added for every vertex *n* in the graph from *s* to *n* where the capacity is equal to the motif degree of *n*. There is also an edge added for every vertex *n* in the graph from *n* to *t* where the capacity is equal to α times the motif size. This flow network structure can be seen in Figure 10 (the edges connecting vertices like 0, u, and v together are implied). As a note, in Goldberg’s original algorithm, it uses edge degree and just α. But CoreExact has adjusted those values to work with motifs other than edges. The time complexity of this construction comes down to the handling of the motifs. Each edge of each motif needs to be handled separately for determining the weight, which is the most complex part of the construction, resulting in a time complexity of . With this graph, we can take the minimum st-cut to see if there exists a subgraph of density α or higher.

A diagram of a graph

AI-generated content may be incorrect.

Figure 10. CoreExact flow network.

Golberg explains the logistics of this min-st cut in this paper [16]. The explanation presented states that for a min st-cut of our constructed graph, if *S / {s} = ∅,* then there is no subgraph of density > α. But if *S* is not empty, then the density of *S / {s} >* α. Goldberg does this by showing the two cases based on his cut capacity equation, where . In this equation *m* is the capacity of edges from *s* to the graph’s vertices, *V* is the set of vertices in *G*, and is *S / {s}*. In these cases, contradictions would arise if was empty or not, proving his theorem. A more through explanation of this is explained in [16], and an explanation for how it’s applied to cliques and motifs is explained in [12].

This process iterates numerous times. With each iteration, we tighten the bounds of *u* and *l*. When α produces a new densest subgraph, *l* is set as α, and otherwise, u is set as *α*. The final key detail here is how many iterations occur. The way it’s set up is that you iterate until , where *v* is the number of vertices in the graph *G*. The logic behind this bounding is that is the smallest possible distance between any two subgraph densities for a given graph, meaning onceis greater than this value, there are no more possible densities to find through the binary search. The proof for this value is that given two densities and , the difference between them is . If , then the difference (since and are interchangeable in this case). But otherwise, the smallest possible difference between and is 1, meaning you have values *v* and *v -* 1 in which case the difference would be [16].

As previously discussed with Goldberg’s algorithm, the time complexity of an implementation of the algorithm was , where *v* is the number of vertices, and *T* is the complexity of the minimum st-cut. After analyzing the CoreExact implementation of Goldberg’s algorithm, this holds true. It is still a binary search depending on the number of vertices *v* (hence log(*v*) iterations), and for every iteration, the most complex step is the minimum st-cut, which is the Edmonds-Karp algorithm. Meaning that the time complexity for the CoreExact implementation of Goldberg’s algorithm comes out to , since we previously covered that the time complexity of Edmonds-Karp was .

## **The Full Algorithm**

Now that we’ve covered the individual aspects, let’s review how this algorithm works. Before the algorithm begins, a graph and motif type are selected as input. And then we run edge-based core decomposition so that we can generate a DAG of the graph, which we then use to generate a list of all motifs in the graph. Which we can then use to run motif-based core decomposition on the graph to find the k-cores and store the information on the densest of them. The densest k-core is then broken into connected components, and the highest density is found between the k-core and its connected components. The densest of these is saved as the current densest subgraph, and we get the upper and lower bounds, *l* and *u*, from them (where *u* = the *k* value of the k-core and *l* = the greatest density). At this point, the binary search for a denser subgraph begins, being run on each of the connected components. The next best guess for highest density is used to construct the flow network. The minimum st-cut is taken of this flow network, returning *S*. We then check if *S = {s}* or not. If *S = {s}* is true, a subgraph of density greater than or equal to α was not found, and thus the upper bound is now set as α. If *S = {s}* is not true, then a subgraph of density greater than or equal to α was found, and the density of that subgraph now equals the lower bound and that subgraph is saved as the current densest. This runs until the difference between the upper and lower bound *u* and *l* are such that . Once that happens, we have found the densest subgraph for that component. We then set the lower bound as the density of that component, and if the density is greater than the density of our current densest subgraph, we set that new subgraph as the densest subgraph. If there is another connected component, we start a new binary search for the densest subgraph of that component with our new lower bound and same upper bound. Otherwise, we are done and have found the densest subgraph.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 11. CoreExact algorithm [2].

For reference in terminology, *p* represents density, *p’’* is the density of the densest k-core, *k’’* is the k value of the densest k-core, ψ is the chosen motif type, (*k’’*, ψ)-core is the densest k-core, *C*’s are components, and *F*’s are flow networks.

Our worst-case time complexity for CoreExact is relatively straight forward to find. We are doing a loop based on the number of connected components, *c*. And for every iteration, the most complex step would be Goldberg’s algorithm. Every other necessary step, including processing our motifs to find the density, is negligible by comparison. So, since we know the time complexity of Goldberg’s algorithm comes out to , and we know it occurs *c* times, we find that the time complexity of CoreExact comes out to.

With all of this in mind, we can cover the important aspects of our implementation of the CoreExact algorithm.

**Chapter 4**

# **Our Parallelized Implementation**

As mentioned before, our solution was implemented and coded with CUDA. Because of the way CUDA works, much of the solution is still serialized and written in C++. So rather than cover all the CUDA code, we will only be covering the parts that were parallelized.

Additionally, while our CUDA solution is mostly an implementation of the CoreExact algorithm, there is one notable change. Rather than using any type of motif, our code is limited to cliques. Cliques are a specific subset of motif, being a subset of vertices that are all connected to one another. They’re often referred to as k-cliques, where the k is how many vertices are in the clique. So, a 3-clique is a clique of 3 vertices, and a 4-clique is a clique of 4 vertices.

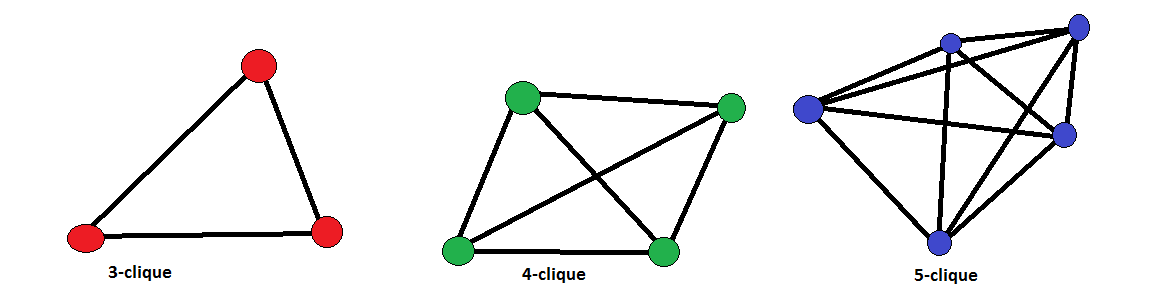


Figure 12. k-clique examples [17].

The CoreExact algorithm and our definition of motif density remain unchanged, with the only thing changing being the process for finding all the cliques. This change was due to not being able to parallelize the listing of any motif types, a process which takes quite a while in the serialized program. Of note, this does not affect our solution’s ability to do edge density, as a 2-clique is an edge by definition (two vertices connected to all other vertices in the clique, which is just each other).

## **DAG Generation**

The first piece of the code to be parallelized is the generation of the DAG. Before this is run, edge-based core decomposition has already occurred.

The first major step is generating new degrees for every vertex to reflect the changes of becoming a DAG (with edges only counting for a vertex’s degree if it has a higher core value than the vertex it’s connected to). This is done using the kernel function generateDegreeDAG, where every vertex is processed by a warp in parallel. The 32 threads of each warp work in parallel to compare the core value of the vertex to its neighbor, adding 1 to a collective degree variable for the warp (and thus the vertex) if the neighbor has a lower order. The structure of this parallelization is shown in Figure 13, visualizing how each warp handles a vertex, and each thread of the warp handles a neighbor of that vertex. This methodology stores the DAG degree since direction is now accounted for in the vertex’s degree, and up to 32 neighbors for every vertex are checked each iteration in parallel. And once all neighbors are checked, the new degree of the vertex is stored.

A diagram of a machine

AI-generated content may be incorrect.

Figure 13. Processing vertex neighbors by thread.

The second major step is setting up the direction of every vertex to reflect the change into a DAG. This is done in a very similar vein as the first step, this time using the kernel function generateNeighborDAG. This follows the structure seen in Figure 13, with every warp processing a vertex, and each thread in the vertex checking a neighbor of the vertex (and thus the edge). And in the case the neighbor has a lower order, the neighbor is stored for this vertex in a new adjacency list. This creates our DAG, storing only the edges in direction of higher order to lower order, with every thread checking a neighbor in parallel.

## **Finding all Cliques**

Just as the algorithm for finding cliques in CoreExact was in depth, so is our CUDA code for finding all cliques in the graph. So, let’s break this down.

The first major step is done with a kernel method called listInitialCliques. This function processes vertices by warp, using its threads to process the neighbors of the vertex. For each neighbor, we check if it’s a valid candidate for a clique, seeing if it has enough neighbors to be part of a clique with the current vertex. This information is stored, creating pairs of vertices that will be checked as being part of larger cliques. So, every thread is working in parallel to check for valid vertex and neighbor pairs that may be part of a larger clique. After this is the kernel method flushPartitions. This function simply updates global information based on current local information, including our current list of potential cliques and neighbors that could extend the clique. This will be used again in a bit.

The next major step is a loop that checks each clique level down to 2, where clique level refers to the size of the clique. Each iteration, the kernel function listMidCliques is called. This function checks all candidate vertices by warp. For each candidate, it checks whether the candidate has valid neighbors for the next clique level. If it does, all valid neighbors of the candidate are added to the list of candidates for the next clique level, and the candidate is officially added to the potential clique it may be a part of. So each thread is checking for valid neighbors in parallel, and each warp is processing candidates. After listMidCliques is finished, we run flushPartitions again to update our global clique information. Then we loop back to the top of the kernel level loop. So in essence, this stage consists of confirming whether candidate vertices can be added to their potential cliques, adding new candidates, and then updating that information to be global after each iteration through the descending clique levels until we get the clique level 2.

The final step uses the kernel method writeFinalCliques. By this point, our potential cliques are all of size *k – 2*, where *k* is the size of our clique. And we have a final list of candidate vertices. So, the potential cliques are all processed by warps, with the threads of each warp checking the neighbors of each of its candidate vertices. For every valid neighbor (a neighbor that is connected to all other vertices in the clique), a complete clique is found. At that point we store the clique (the vertices in the clique) and update the clique degree of all the vertices in the clique. So, every thread is checking for potential complete cliques in parallel. Afterwards, the list of cliques and the clique degrees of the vertices are stored globally.

## **Clique-based Core Decomposition**

And now we get to what we’d consider the first step of the CoreExact algorithm, being core decomposition based on the chosen clique size. The parallelization here occurs inside a serial loop, with each iteration checking a core level, a different *k* value. So, the first iteration checks the 0-core, the second iteration is the 1-core, and so on. This runs until all vertices have been peeled from the graph, and we keep track of the densest k-core by storing data on the densest k-core each iteration, updating it when a denser k-core is found.

Each iteration, two main steps occur in parallel. The first is done with the kernel method selectNodes. Every vertex is processed on an individual thread, meaning as many vertices are being processed at the same time as there are threads. The parallel structure here is simple, which is visualized in Figure 14. Each thread checks whether the degree of that vertex is equal to the current level (our *k* value). If it is equal, that means it is part of the current k-core, and therefore it is added to a buffer of vertices that need to be removed by the next level (we’ll call this the removal buffer).

A line with black text

AI-generated content may be incorrect.

Figure 14. Processing vertices by thread.

The second step is done with the kernel method processNodesByWarp. In this method, we process every vertex in the removal buffer by warp, and each thread of the warp processes a neighbor of the vertex. So, for every vertex in the removal buffer, we check every clique that the vertex is a part of. For each neighbor in the clique, we decrease the clique degree of the neighbor by 1. In the case that this decrements that vertex to have a degree equal to our current level, that vertex gets added to the removal buffer to be handled next iteration. In the case that the decrement brings that vertex’s degree below the current level, we increment back up by 1, as it would already be in the buffer or it would have already been peeled. This makes it so that at the end of this, the array holding the degrees of every vertex will instead hold the core value of every vertex.

After these two parallelized processes have run, we increase our level by 1 so we can check the next k-core in the next iteration, and then we check the density of our current k-core. If the current k-core’s density is greater than the currently stored greatest k-core density, we update that densest k-core information to be the current k-core.

## **Construction and Pruning of the Densest k-Core**

With the information about the densest k-core gathered, in the code, we need to construct the densest k-core subgraph so that we can work with it rather than just know what vertices are in it. So, we need to generate a graph representative of the densest k-core and relevant information about the graph. This process is done in two major steps with two kernel functions.

First up is generateDensestCore which handles the bulk of the construction. This function handles vertices by warp. For each warp, we check if the vertex has a clique core value greater than or equal to that of the densest k-core, confirming whether the vertex is part of the densest k-core. If it is, we add it to our new ongoing list of vertices in the subgraph. Once it’s added, we check all its neighbors in parallel, with each thread of the warp checking one neighbor at a time. For every neighbor that also has a clique core value greater than or equal to the densest k-core value, we increase clique degree of the vertex by 1 (with it starting at 0 for this new subgraph). Meaning by the end of this parallelized function, we have the set of all vertices in the densest k-core subgraph and all of their clique degrees.

The second step of this process is done with the kernel function generateNeighborDensestCore. As keeping with our adjacency list structure for the graph, the other major piece of information we need to store for the graph is what vertices each vertex is connected to in the densest k-core (we need to exclude all neighbors outside of the densest k-core). Like the previous step, we are handling our vertices by warp, and each neighbor is being handled by the threads in these warps. For each thread, we check one of the warp vertex’s neighbors, checking to see if it is part of the densest k-core. If it is, we add it to the array of neighbors of the vertex.

And with these two steps, we have stored the list of vertices in the subgraph, the clique degree of each vertex, and all their neighbors in the subgraph. However, we’re not quite done. Now we must prune our graph for unnecessary edges. That isn’t to say that there are edges in this subgraph that aren’t part of a clique. But as we are looking for the densest subgraph, there’s a chance certain edges aren’t part of *enough* cliques to be relevant. That is to say, we want to find and eliminate all edges of low density. This will be done in two main steps, identifying the edges to remove and updating the clique degrees and vertex neighbors.

To identify edges to remove, we will instead do the opposite and identify all edges that need to stay. So, in our kernel function pruneEdges, we handle all the cliques we’ve identified by warp. For every clique, we check if its clique value is greater than the densest k-core value (e.g. confirming a 5-clique is part of our 4-core but a 3-clique is not). For every clique that is part of our k-core, we mark all its edges as necessary over the warp’s thread in parallel. So, any edge that never gets marked is not part of a large enough clique is not necessary and will be removed. This parallelization is visualized in Figure 15.

A diagram of a diagram

AI-generated content may be incorrect.

Figure 15. Handling cliques by warp and its edges by thread.

Once our pruning identification is done, we can effectively prune the unnecessary edges by updating the clique degrees and neighbor lists of each vertex. This is done in two kernel functions, generateDegreeAfterPrune and generateNeighborAfterPrune. These work in a very similar way. Both handle each vertex in a warp and check each neighbor by thread in the warp. For each neighbor, we check the edge’s removal status. As we covered, we mark every edge that needs to be removed. In generateDegreeAfterPrune, we keep a count of every neighbor that isn’t marked for removal. Then once every neighbor is processed, we set the clique degree of the warp vertex to be that count as it is all neighbors unpruned. In generateNeighborAfterPrune, we keep a list of every neighbor that isn’t marked for removal. Then once every neighbor is processed, we set the neighbor list of the vertex as this new list. Meaning that these two functions process the neighbors of every vertex in parallel, keeping track of what neighbors aren’t being removed to update the degree and neighbor list of each vertex. And once these two functions are run, we now have our densest k-core subgraph with all of its relevant information stored.

## **Component Decomposition**

The next parallelized section is the component decomposition of the densest k-core. To start this off, we will store component ids for every vertex, with every id simply starting as the vertex id in an array, so vertex numbers align with the index numbers of the array. Using these ids, we can narrow down our components by finding the minimum component id that every vertex is connected to.

This leads us to our main loop, a do while loop that runs until no component id changes occur, kept track of using a boolean value named “changed”. For each iteration, we run the kernel function componentDecomposek. In this function, every vertex is handled by warp. For each warp, we store the current component id of the vertex and set it as our minimum component id, then we start checking the vertex’s neighbors over each thread in the warp. For each neighbor, if the neighbor’s component id is less than the current minimum id, then the neighbor’s component id is set as the minimum. After every neighbor is checked, if a minimum id was found that was less than our vertex’s, we set the vertex’s new component id as the minimum, and update “changed” to be true. So in parallel, we continually check to see if each vertex has a neighbor of lower component id, updating the vertices to be in the lowest component id they are connected to by any path in the graph, until no more updates to component id occur.

Then, utilizing the C++ parallel functions sequence and sort\_by\_key which run on the GPU, we can store our connected components. The sequence function simply creates an array of numbers from two given numbers as the range (e.g. given 3 and 7 will create an array storing the numbers 3, 4, 5, 6, and 7). In this case, we use it to create an array of all the vertices in our densest k-core. Sort\_by\_key is given 2 arrays, a list of keys (in our case our component ids), and our values (our vertices). This function sorts the keys and matches them with the values. So, all the component ids are sorted, then the vertices they match to are sorted to match the new order of the component ids. This way, we have our components stored as a pair of linked arrays.

Additionally, we store two final pieces of info using two more C++ parallel functions. First, by running the function unique\_copy on our list of component ids, we find every unique component id and store that number, effectively counting how many connected components there are. Then, we use the function lower\_bound to create an array of every index in the array of component ids that represents the start of a component. Of note, the parallel C++ functions utilized for creating our linked arrays and storing the connected component information are from the “thrust” library. And with that, our connected components are found and can be worked with individually.

## **Bound Generation**

As we know, for Goldberg’s algorithm we need an upper and lower bound for the density being checked so that we can work through in a binary search. In CoreExact, we used the same upper bound each time and updated the lower bound after each iteration. However, to parallelize the process we will avoid relying on previous iterations for lower bounds and instead generate an upper and lower bound for each connected component.

The upper and lower bounds will be stored in a pair of arrays, with their indices linking them in representation of the component. Next, just like in CoreExact we also store the maximum k-core value to serve as a strict upper bound for all of the components. Then we run the kernel function getLbUbandSize. Each component is handled on a thread. This parallelization is depicted in Figure 16. For each component, the lower bound is set as the clique density of that component. Then, for upper bound, we find the maximum possible number of k-cliques for the component (a theoretical fully connected graph) and divide by the number of vertices in the component. Whichever value is higher between the fully connected component density and the maximum k-core value is stored as the upper bound. Finally, if the lower bound is greater than the upper bound, we discard the component entirely, not storing it in an ongoing array. Otherwise, we do store it in this array. And with that, the upper and lower bounds for each connected component is calculated in parallel allowing us to process each component in parallel rather than sequentially.

A diagram of a line with letters and numbers

AI-generated content may be incorrect.

Figure 16. Handling components by warp.

## **Flow Network Creation**

With our bounds set, we can now create the unique flow networks necessary for finding the densest subgraph for each component. As a reminder, the structure of these flow networks is a source and sink node connecting to every vertex in the graph, with edges in the graph all having capacities of 1, edges from the source to each vertex with a capacity of the vertex’s motif degree (which will now be clique degree), and edges from the vertices to the sink with a capacity of α times motif size (where α is the average of the upper and lower bound and motif size is now clique size).

This will be parallelized across our various connected components. And the first step is simply allocating memory for the edges of our flow networks. Using the kernel function createFlowNetworkOffset, our components are handled by warp. For each component, memory offsets are created according to the number of edges that need to be stored. This is done by handling the vertices in a parallel, with each vertex running on a thread, setting the number of edges for that vertex as the clique degree of the vertex. This parallel structure is depicted in Figure 17. Then the source and the sink have edges set up as the number of vertices in the graph since each needs to connect to each vertex.

A diagram of a diagram

AI-generated content may be incorrect.

Figure 17. Handling components by warp and their vertices by thread.

And now with our offsets created and memory allocated, we can fill in the flow network information. Like the previous function, this is done with each component being handled by warp, and each vertex being handled by thread as seen in Figure 17. The flow network is handled in a structure with two linked arrays, with one storing the edges and the other storing the capacity of the edge it’s linked to. First, every edge from a component vertex to the sink is set, with their capacity being equal to α times the clique size. And then, every edge from the source to a vertex is stored, with their capacity being equal to the clique degree of that vertex. So in parallel, every edge of the flow network and its capacity is set in parallel across all the threads of their component’s warp.

## **In Summary**

Of the CoreExact algorithm, our solution has parallelized DAG generation, clique listing, clique-based k-core decomposition, densest k-core construction and pruning, component decomposition, bound generation, and flow network creation. Not every part of CoreExact was parallelized into our program. That was due to time constraints and certain parts not being able to run in parallel. For example, running a minimum st-cut algorithm or the bound tightening loop of Goldberg’s algorithm in parallel is not possible due to the consecutive dependencies of the process. While a process like updating the flow networks based on new upper and lower bounds is potentially possible but will have to be pursued in an updated version of this code due to time constraints. However, as it stands our solution parallelizes most of the code, leaving only the main loop with sections to be parallelized. And it is this version of the code we will run efficiency tests on.

**Chapter 5**

# **Experiment and Evaluation**

(continue here)

# **References**

|  |  |
| --- | --- |
| [1] | C. M. Y. F. a. L. V. L. W. Luo, "A Survey of Densest Subgraph Discovery on Large Graphs," 14 June 2023. [Online]. Available: https://arxiv.org/abs/2306.07927. [Accessed 6 January 2025]. |
| [2] | K. Y. R. C. L. V. L. X. L. Y. Fang, "Efficient Algorithms for Densest Subgraph Discovery," 17 August 2019. [Online]. Available: https://arxiv.org/abs/1906.00341. [Accessed 6 January 2025]. |
| [3] | Udacity, "Intro to Parallel Programming," 5 August 2013. [Online]. Available: https://www.youtube.com/watch?v=F620ommtjqk&list=PLAwxTw4SYaPnFKojVQrmyOGFCqHTxfdv2. [Accessed 6 January 2025]. |
| [4] | A. M. A. F. F. B. T. Lanciano, "A Survey of Densest Subgraph Discovery Problem and Its Variants," 18 April 2024, Apr. 18. [Online]. Available: https://arxiv.org/abs/2303.14467. [Accessed 8 January 2025]. |
| [5] | G. K. a. D. P. U. Feige, "The Dense k-Subgraph Problem," in *Algorithmica*, New York, Springer-Verlag, 2001, pp. 410-421. |
| [6] | R. K. a. V. Vinay, *Analyzing the Strucutre of Large Graphs,* August, 1999. |
| [7] | M. Charikar, "Greedy Approximation Algorithms For Finding Dense Components in a Graph," *Proceedings of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization,* pp. 84-95, 2000. |
| [8] | F. B. A. G. F. G. a. C. Tsourakakis, "Denser than the densest subgraph: extracting," *KDD,* p. 104–112, 2013. |
| [9] | G. Shute, "University of Minnesota Duluth," Fall 2014. [Online]. Available: https://www.d.umn.edu/~gshute/cs2511/slides/graphs/slide017.html. [Accessed 2 February 2025]. |
| [10] | P. S. Segundo, "ResearchGate," June 2010. [Online]. Available: https://www.researchgate.net/figure/A-A-simple-undirected-graph-U-B-The-adjacency-matrix-A-of-U-C-Bit-wise-AND-operation\_fig6\_220204855. [Accessed 2 February 2025]. |
| [11] | R. U. Orakzai, "What is the K-Core of a Graph," Baeldung, 5 June 2023. [Online]. Available: https://www.baeldung.com/cs/graph-k-core. [Accessed 5 February 2025]. |
| [12] | J. P. R. P. C. T. S. C. X. M. Mitzenmacher, "Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling," *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* pp. 815 - 824, 2015. |
| [13] | J. Kogler, "Maximum Flow - Ford-Fulkerson and Edsmond-Karp," Algorithms for Competitive Programming, 13 October 2024. [Online]. Available: https://cp-algorithms.com/graph/edmonds\_karp.html. [Accessed 10 February 2025]. |
| [14] | L. Mouatadid, "Network Flows: The Max Flow/Min Cut Theorem," University of Toronto, Toronto, 2016. |
| [15] | "DSA Edmonds-Karp Algorithm," W3 Schools, [Online]. Available: https://www.w3schools.com/dsa/dsa\_algo\_graphs\_edmondskarp.php#:~:text=So%2C%20BFS%20with%20time%20complexity,(%20V%20%E2%8B%85%20E%202%20)%20.. [Accessed 11 March 2025]. |
| [16] | A. V. Goldberg, Finding a Maximum Density Subgraph, Berkley: Univerisity of California, 1984. |
| [17] | "Graph kclique\_percolation," 8 January 2016. [Online]. Available: https://pythonhosted.org/trustedanalytics/python\_api/graphs/graph-/kclique\_percolation.html. [Accessed 14 April 2025]. |
| [18] | "Introduction to Push Relabel Algorithm," GeeksForGeeks, 6 June 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-push-relabel-algorithm/. [Accessed 4 May 2025]. |
| [19] | M. Aibin, "Connected Components in a Graph," Baeldung, 18 March 2024. [Online]. Available: https://www.baeldung.com/cs/graph-connected-components. [Accessed 5 February 2025]. |
| [20] | S. Y. Cheung, "The Maximum flow and the Minimum cut," Emory University, 27 October 2019. [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/NetFlow/max-flow-min-cut.html. [Accessed 10 February 2025]. |