Evaluating and Improving Performance in a GPU Graph Framework

By

ANNIE ROBISON

PROJECT REPORT

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Matthew K. Farrens

Jason Lowe-Power

Committee in Charge

2024

CONTENTS

List of Figures

LIST OF TABLES

ABSTRACT

Evaluating and Improving Performance in a GPU Graph Framework

Analysis of instrumentation data can be used to improve performance in a GPU graph framework. In this paper, we present our design goals and implementation of performance analysis in Gunrock 2.0. We discuss various competing design goals and how they impact implementation. We then conduct performance analysis on our MST algorithm and optimize the algorithm based on this analysis.

Chapter 1

Introduction

Graphs and graph algorithms have a number of applications spanning various industries. One industry that heavily employs graphs is gaming. The possibilities of how graphs could be utilized in games are as limitless as an open-world RPG. A graph could be used to map different quest lines a player could take or to store connections and distances between different locations on a game map. Graph applications, though, are not all fun and games. Graphs could also be used to map terrorist networks. An algorithm could take in such a graph and use it to predict who is likely to have information on an upcoming attack. Graph algorithms are also utilized in manufacturing, fraud detection, and forensics to name a few.

As many of these applications are time-sensitive, the performance of graph algorithms matters. The real-time nature of games, which can render one hundred frames per second or more, makes strong performance non-negotiable. In matters of national security, weak performance could cost lives.

Instrumentation–the ability to measure performance–can be key to improving performance. Measuring performance, however, can be difficult and costly. Many competing design goals exist. These include ease of use, ease of development, minimizing overhead with instrumentation on, and minimizing overhead with instrumentation off. The designer

must weigh these goals, and ensure the implementation lines up with the chosen priorities. Moreover, instrumentation code itself can hurt performance. Actions must be taken to counteract this.

Though studies such as Xu et al. [4] also look at the performance of GPU graph algorithms, this paper uniquely explains how to implement and conduct performance analysis in this environment.

We present our performance evaluation design and discuss different design approaches based on different goals. This will aid others implementing instrumentation in similar environments. We also walk through performance analysis and optimization on an example algorithm to show how to conduct analysis and optimize algorithms after the instrumentation system is in place.

Chapter 2

Background

2.1 Performance Analysis Example

To lay a foundation for the rest of this paper, let us first look at how we could do performance analysis with breadth first search on an example graph. Using Jane Austen's *Pride and Prejudice*, we will construct a graph made up of various love interests throughout the novel. These are illustrated in Figure 2.1.

At the end of the novel, Elizabeth Bennet marries Mr. Darcy. Their mutual interest in one another is indicated with a double-sided arrow. Elizabeth, however, never requites Mr. Collins' affection; the arrow from Mr. Collins to Elizabeth is one-sided. If you're familiar with *Pride and Prejudice*, you will already know this illustration is simplified. It includes only those whose trails of affection somehow connect to Elizabeth Bennet. It does not even mention Jane and Mr. Bingley! One can imagine how complex graphs can become. To further complicate matters we could assign a number, or weight, to each character's affection for another. Lydia Bennet's affection for Mr. Wickham, for example, would unfortunately have a higher weight than his for her.

Breadth first search (BFS) is a common graph algorithm which iterates over the graph to search it one level at a time. Figure 2.2 shows a BFS on the graph from Figure 2.1,

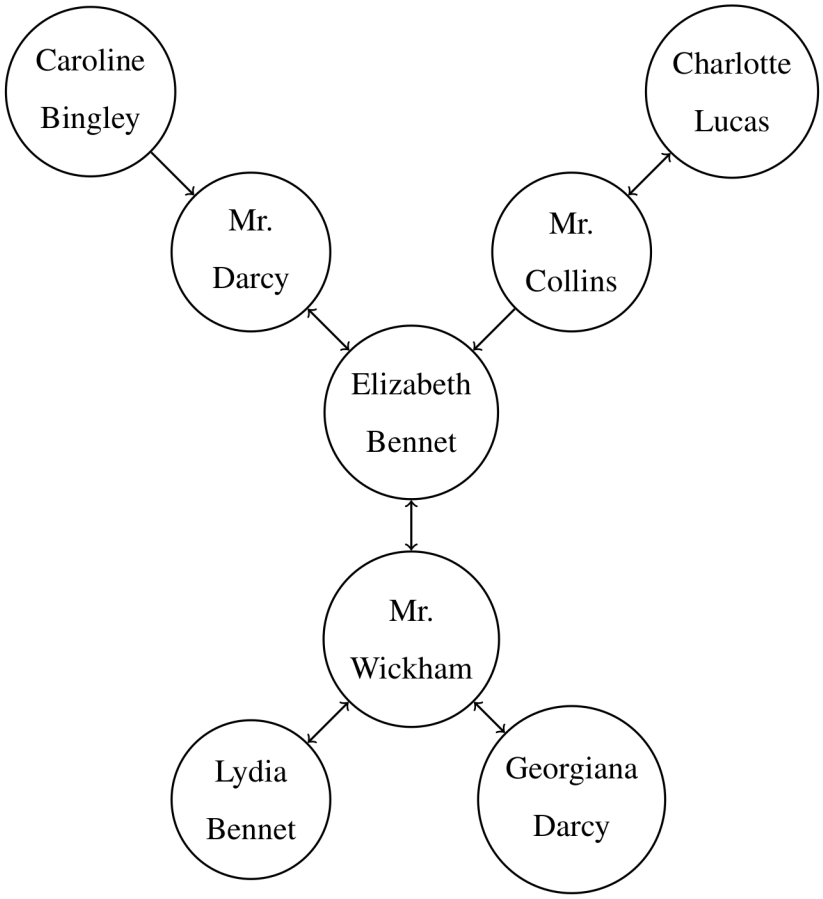

Figure 2.1: Love interests in *Pride and Prejudice*

starting with Elizabeth Bennet. The BFS algorithm would first visit Elizabeth's love interests (Mr. Darcy and Mr. Wickham), then their unvisited love interests (Lydia Bennet and Georgiana Darcy).

Some metrics we could collect when conducting performance analysis on this algorithm and graph include edges visited, vertices visited, search depth, and runtime. We will discuss these metrics and their importance in more detail in the following chapter. The edges visited here would be four; first we visit the edges between Elizabeth and her two love interests, then the edges between Mr. Wickham and his two other love interests.

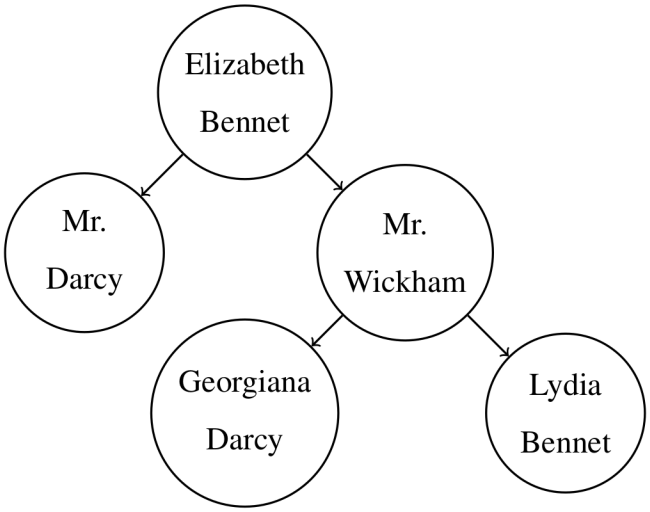

Figure 2.2: A BFS of the graph in Figure 2.1

We visit each vertex once so vertices visited would be five. There are two iterations: one starting at Elizabeth Bennet and another starting at Mr. Darcy and Mr. Wickham. Thus the search depth is two.

2.2 About Gunrock

The Gunrock framework [3] was designed to simplify the process of developing high-performance GPU graph algorithms. Gunrock's built-in operators do the heavy lifting on the GPU. A user can employ one or more operators to construct their implementation. BFS, for example, utilizes the advance operator. Each advance operation takes in a subset of the graph (a frontier) and outputs a new frontier consisting of its neighbors. The initial frontier in our BFS above would consist of Elizabeth Bennet. Then it would advance to Elizabeth's love interests, outputting a frontier of Mr. Darcy and Mr. Wickham. This is an iterative process; multiple advance operations would take place in a loop until the algorithm converges to a solution.

One of the most difficult aspects of programming efficient graph algorithms on the

GPU is load balancing. Because operations like advance are inherently unbalanced (some nodes will have more neighbors than others), this presents a challenge in how to distribute work on the GPU. One of Gunrock 2.0's main advantages is that it includes powerful load balancing capabilities. The programmer can specify a load-balancing scheme, but the details are abstracted away; thus developers do not need to worry about these low-level GPU optimizations and can focus on their algorithms.

Chapter 3

Design

3.1 Metrics

A key part of designing a performance evaluation system is deciding what metrics to collect. In the case of GPU graph algorithms, we want to know how quickly and efficiently an algorithm is able to traverse graphs. Our metrics should reflect this.

Basic but important nonetheless, **runtime** measures how long the algorithm takes to complete. Tracking the numbers of **vertices and edges visited** gives a sense of how much total work the algorithm is doing. In our implementation, these two metrics are the most costly to collect. We will discuss the reasons for this in the following sections. Yet, the cost is justified as they are vital to understanding the algorithm's performance. Together with runtime, they also provide a measure of throughput. The metric commonly used for throughput is millions of edges traversed per second (**MTEPS**). This is calculated using edges visited (in millions) divided by runtime (in seconds). Finally we have **search depth**: the number of iterations the algorithm takes to converge.

3.2 Design Goals

The audience and emphasis of a framework will impact instrumentation design goals. Four major design goals exist: minimizing overhead with instrumentation off, minimizing overhead with instrumentation on, ease of development, and ease of use.

If minimizing overhead when off is a design goal, there should be no interference from performance evaluation code when it is turned off. This should be a goal for most if not all GPU graph frameworks. What good is performance analysis if it itself hurts performance? The most effective way to eliminate this overhead is through preprocessing. Every piece of instrumentation code should be wrapped in a conditional that is evaluated at compile time and checks if instrumentation is active. Any variable checked at runtime will have the potential to impact performance. Of course, this method comes with costs. The user will need to compile and run separate executables with performance evaluation on and off. Runtimes must be collected with performance evaluation off, yet other metrics must be collected with it on. This complicates the process of compiling results. Thus if ease of use is a design goal, this may not be the right choice.

Another way to accomplish this is to maintain two separate copies of the program: one with performance analysis enabled and one with it disabled. A runtime conditional would check if it is enabled or disabled and call the appropriate program. Because the runtime check would occur at the outermost level, it would only be called once and the performance impact would be minimal. While effective, this method is taxing for developers. Writing one program is more than enough work! This is not the right option if ease of development is a priority.

If minimizing overhead with instrumentation on is a priority, every algorithm would implement its own performance analysis. This is because the most efficient way to gather performance data will vary from algorithm to algorithm. For some algorithms, one can

simply find the size of the frontier in each iteration and add it to a variable to track edges or vertices visited. However, this will not work for all algorithms as not all algorithms utilize frontiers in the same way. Prioritizing performance with instrumentation on is important in systems that conduct constant performance analysis. However, this goal directly competes with ease of development. The burden of implementing instrumentation is placed on each algorithm's developer.

Ease of development is an important goal for open source frameworks. If it is a priority, all instrumentation should take place at the framework level and not the algorithm level. This will unfortunately lead to greater overhead when performance analysis is enabled.

Ease of use is also important in systems where performance analysis will be run frequently. If ease of use outweighs ease of development and minimizing overhead when off, runtime conditionals checking if instrumentation is enabled may be acceptable. Conditional checks should occur at the outermost layer possible. Checks that are deeper into the program and need to be called more than once have a greater potential to impact performance and should be avoided. Runtime conditionals should especially be avoided on the GPU. Otherwise, each thread will have to pay the cost of the conditional check.

In Gunrock 2.0, we prioritize ease of development and minimizing overhead with instrumentation disabled. These goals directly correlate to the goals of the framework itself. Gunrock aims to be a framework that researchers and developers can use to develop novel graph algorithms easily. Adding the burden of individually implementing instrumentation would violate this goal. Moreover, Gunrock 2.0 is a high-performance GPU graph analytics framework (emphasis on high-performance). Allowing instrumentation code to detrimentally impact performance would also go against the goals of our framework.

3.3 Implementation

With these goals in mind, we implement performance analysis in Gunrock 2.0. We add the following functionality: tracking edges visited, vertices visited, runtime, and search depth; exporting metrics to JSON files; and controlling instrumentation with an on/off switch. Our design decisions reflect the design goals of minimizing overhead with instrumentation off and enabling ease of development. As such, we use preprocessing for our on/off switch and add our instrumentation code at the framework, not algorithm, level.

We define a CMake variable indicating if performance evaluation should be turned on or off. With it on, the program defines and initializes the relevant data structures. Then, it atomically increments counters within the operator code itself. Because the conditionals are evaluated at compile time, the branch not taken is not included in the compiled executable. With this method, there is no impact on performance when metrics collection is turned off. The main shortfall is that separate executables are needed to collect runtimes and the other metrics. To remedy this, we use a wrapper script to run both programs and combine their results.

To use a single runtime conditional with separate programs, the amount of extra code we'd need to write and maintain would be tremendous. Not only would we need to have two separate versions of each algorithm (or its main function body), we'd also need separate versions of each of our operators: one that calls to increment our counters, and one that does not. The separate versions would be needed because runtime conditional checks within these functions would severely impact performance. Writing and maintaining multiple versions of our code would violate ease of development. Of course, if instrumentation was implemented at the algorithm level instead, multiple operator implementations would not be needed. However, such an implementation would also violate ease of development, as each algorithm's programmer would be responsible for implementing performance anal-

ysis. Our implementation was chosen because it simultaneously meets both of our chosen design goals.

A benchmark class contains most of the relevant code. It holds the edges visited, vertices visited, search depth, and total runtime. It also includes a data structure with device pointers for edges visited and vertices visited, since these metrics must be incremented in device code. To facilitate this process, it defines functions to increment these values using atomic addition. When an operator (such as advance) visits an edge or vertex, it calls these functions. The code injected into the operator is of course also wrapped with a preprocessing conditional and thus not included when performance evaluation is turned off.

Prior to starting a run of an algorithm, a function is called to initialize the benchmark data structures, regardless of whether performance analysis is on or off. With performance analysis off, all code in this function is filtered out of the compiled executable so nothing happens when it is called. After the run has completed, a function is called to extract the data from the GPU to the host and return a struct with the metrics. If performance evaluation is off, it simply returns a struct with the default values. Finally, a function is called to destroy the benchmark data structure as all metrics must be reset between runs. Again, nothing occurs if performance evaluation is turned off.

This benchmark struct later feeds into a function that outputs a JSON file with all the pertinent data. This function checks the preprocessing boolean and, if metrics collection is enabled, outputs runtimes, all benchmark data, and reproducability information. If it is not enabled, it simply outputs the runtimes and reproducability information.

We start the runtime timer when the algorithm's first loop begins and stop it when the final loop completes. We do not include the time to read in the data and transfer it to the GPU. This process is independent of the algorithm. The focus of this paper is GPU graph algorithm instrumentation that enables developers to improve their algorithms. Since our

analysis is focused on the algorithm itself, that is the runtime we collect. Data reading and transfer time should certainly be optimized as well, though it is outside the scope of this paper.

We randomly select source nodes for applicable algorithms. However, a random source could be from a small connected component that is not truly representative of the graph. This could skew results. Thus we throw out any runs with runtimes more than two standard deviations away from the mean.

Variables we include for reproducability include GPU info (name, clock rate, driver version, etc.), system info (OS, instruction set, node name, etc.), time, compiler, and Git commit hash. These are all included in the JSON output file.

Figure 3.1 underscores the need for an on/off switch in performance evaluation. It shows the average MTEPS of BFS runs across various SuiteSparse datasets [1] when performance evaluation is on vs off. The runs were conducted on a Tesla V100 GPU. The scatter plot was fitted with both linear and polynomial regressions. In every case, the average MTEPS is higher with performance evaluation turned off and the slope on the linear regression line is only 0.6. Though the linear trendline fits the data well with an R^2 of 0.96, polynomial regression catches a pattern in the data that the linear trendline misses. It provides an even better fit with an R^2 of 0.98.

As captured by the polynomial trendline, the performance hit is larger on average for datasets with higher MTEPS. For the hollywood-2009 dataset, which has the highest MTEPS, MTEPS with performance analysis off is nearly double the value with it on. On the other hand, for the asia-osm dataset, which has the lowest MTEPS, the performance hit is only 1%. One reason graphs with higher MTEPS are more impacted is that their high level of parallelization makes them more likely to suffer contention when any serialization is introduced. Every time an edge is visited, we atomically increment our counters. Higher MTEPS means more edge accesses over a period of time, and thus more contention for

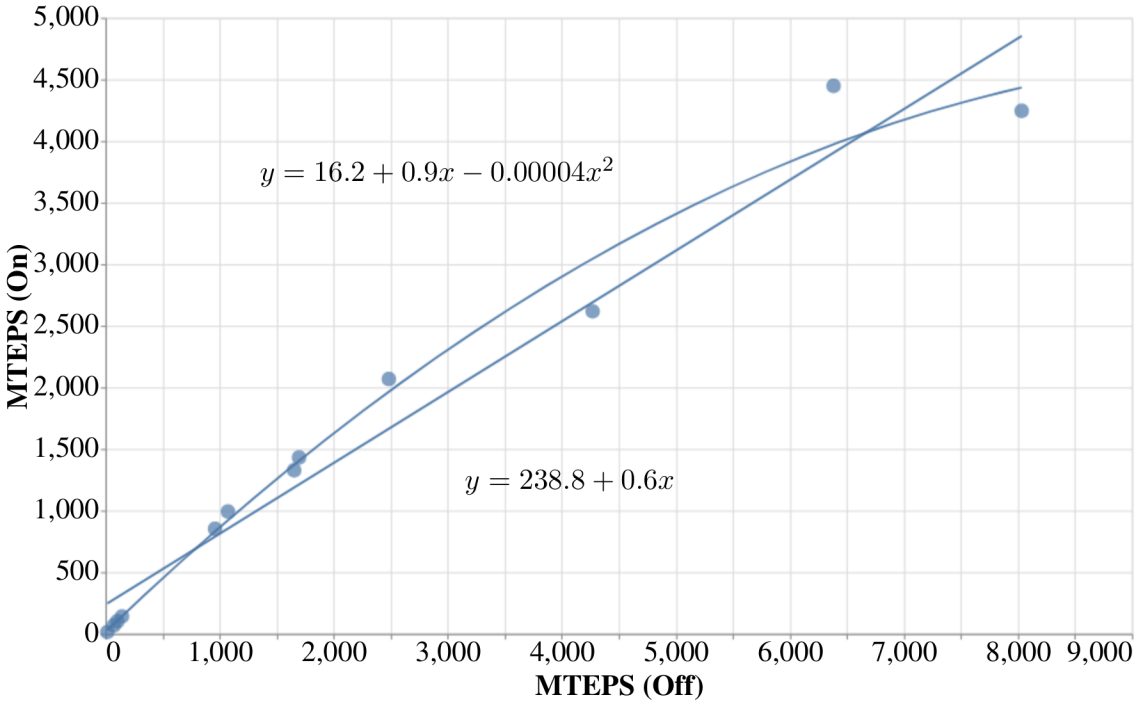

Figure 3.1: BFS MTEPS with performance evaluation on vs off; Tesla V100 GPU; various datasets

that atomic operation. This increases the cost of incrementing our counters.

To demonstrate this, we conducted an experiment where we replaced the atomic operators in our benchmark code with non-atomic operators. Though doing so made the counters inaccurate, we used the values from the original experiment combined with the updated runtimes to observe the effect on MTEPS. This change improved performance with instrumentation on by 15% on average across the top six datasets ordered by MTEPS. For hollywood-2009, it led to a 33% performance gain. However, this change only improved performance by 1% on average for the other six datasets. This experiment shows the greater impact of contention on performance decline for datasets with higher MTEPS due to the use of atomic operators.

Chapter 4

Case Study: Minimum Spanning Tree

Minimum Spanning Tree (MST) finds a tree that connects all vertices in a graph with a minimum weight. It is commonly used for network design with applications from computer networks to electrical grids. It is also used to approximate a solution for the NP-hard Travelling Salesman Problem [2]. In this section we will look at our sequential MST algorithm and how we adapted it for the GPU. We will then conduct performance analysis on our implementation and use this analysis to optimize it. Finally, we will compare our optimized algorithm to the original.

4.1 Algorithm

Our MST algorithm is based on Boruvka's algorithm. Boruvka's algorithm was originally published in 1926 as a method for minimizing the cost of electrical networks. It begins with every vertex as the root of its own tree. Then in each iteration, the outgoing edge with the minimum weight for each tree is added to the MST. As two smaller trees are combined with the addition of an edge, the root of one tree is updated to the root of the other. The process continues until only one root remains, and all vertices are connected in one tree.

4.2 Sequential

Our sequential implementation of Boruvka's algorithm, shown in Algorithm 1, directly correlates to the process described above. Still, a closer look at the sequential algorithm will set the stage for the GPU adaption. After initializing variables, we iterate over all edges to find the minimum weight outgoing edge for each root, storing the minimum weight and the associated neighbor vertex. Then, we iterate over the vertices again check if the vertex has a minimum weight set (that means it is the root of a tree). If it is, we next check if the roots have changed for the root and the minimum neighbor; root changes from adding other edges to the MST would not yet have propagated to all vertices. After these updates, we check that the two vertices are not already connected. If not, we add the minimum weight to our MST weight, update the root to minimum neighbor's root, and decrement our number of trees. All of this takes place within a while loop that continues as long as the number of trees is greater than one.

Algorithm 1 Gunrock Sequential MST Algorithm

Input: Graph $G = (V, E)$
Output: Float w MST weight

```
1:  procedure GUNROCKSEQUENTIALMST(G)
2:      ▷ Initialize MST weight to 0
3:      Initialize w ← 0
4:      ▷ Initialize the number of trees to the number of vertices
5:      Initialize trees ← |V|
6:      ▷ Initialize the root of each vertex to itself
7:      Initialize R ← v∀v = v
8:      ▷ Stop when all vertices are connected
9:      while trees > 1 do
10:         ▷ Assign max value as the min weight for each vertex
11:         Initialize W ← v∀v = numericLimit
12:         ▷ Assign max value as the min neighbor for each vertex
13:         Initialize N ← v∀v = numericLimit
14:         ▷ For each vertex in the graph
15:         for v ∈ V do
16:             ▷ For each edge starting with v
17:             for edgee ∈ edges(v) do
18:                 ▷ Set u to the destination vertex of e
19:                 u = destination(e)
20:                 ▷ If u and v have different roots and u < v
21:                 if R[u] ≠ R[v] & u < v then
22:                     ▷ If weight is less than the current min of v's root
23:                     if weight(e) < W[R[v]] then
24:                         ▷ Set the min weight and min neighbor
25:                         W[R[v]] = weight
26:                         N[R[v]] = u
27:                     end if
28:                     ▷ If weight is less than the current min of u's root
29:                     if weight(e) < W[R[u]] then
30:                         ▷ Set the min weight and min neighbor
31:                         W[R[u]] = weight
32:                         N[R[u]] = v
33:                     end if
34:                 end if
35:             end for
36:         end for
37:         ▷ For each vertex in the graph
38:         for v ∈ V do
39:             ▷ If v is a root
40:             if W[v] ≠ numericLimit then
41:                 ▷ Set u to v's min neighbor
42:                 u = N[v]
43:                 ▷ Update v's root and its min neighbor's root
44:                 updateRoots(v)
45:                 updateRoots(u)
46:                 ▷ If u and v are not already connected
47:                 if R[v] ≠ R[u] then
48:                     ▷ If v < u or the reverse edge is not included
49:                     if v < u or N[R[u]] ≠ v then
50:                         ▷ Add weight to the MST weight
51:                         w+ = W[v]
52:                         ▷ Set v's root to u's root
53:                         R[v] = R[u]
54:                         ▷ Decrement trees
55:                         trees − −
56:                     end if
57:                 end if
58:             end if
59:         end for
60:     end while
61: end procedure
```

4.3 Parallel

Algorithm 2 shows an overview of our GPU adaption of Boruvka's algorithm. In the while loop, we first visit all edges in parallel and use atomic min to find the minimum weight of all outgoing edges for each root (*getMinWeights*). This function outputs a frontier with all edges that were at some point a minimum. Next, we check all edges of the output frontier in parallel and find the edge for each root that has the lowest index of all of its minimum-weight edges (*getMinNeighbors*). Then, we visit all vertices in the graph to add the minimum outgoing edge for each tree to the MST in parallel (*addToMST*). Finally, we visit each vertex in parallel in order to propagate root updates (*updateRoots*).

The *getMinWeights* function is shown in Algorithm 3 and operates on each edge of the graph. It first checks that the source and destination are not already part of the same tree, and that the source is less than the destination. The latter check allows us to operate on two edges (an edge and its reverse) at once. The function then uses *atomicMin* to atomically check if the weight of this edge is less than the current minimum for either root, and update the minimum if so. If a new minimum was found for either root, the function returns true to keep the edge in the frontier. Otherwise, it returns false to remove the edge from the frontier.

Algorithm 4 shows the *getMinNeighbors* function. It operates on each edge in the output frontier generated from *getMinWeights*. This means each edge in the frontier was a minimum *at some point*. An edge could have been a minimum but later been beaten by a different edge and still be in the frontier. The function compares the edge weight to the final minimum for the roots of the source and destination to check that it is a true minimum. If it is, atomicMin atomically checks if the edge has the current minimum index of all such edges for the root and updates the value. This atomicMin check allows us to break ties while also preventing loops. Because of the consistent ordering, two neighboring trees

Algorithm 2 Gunrock Parallel MST Algorithm

Input: Graph $G = (V, E)$
Output: Float w MST weight

procedure GUNROCKMST(**G**)
  ▷ Initialize MST weight to 0
  Initialize $w \leftarrow 0$
  ▷ Initialize frontier F to all edges in G
  Initialize $F \leftarrow E$
  ▷ Initialize number of trees to number of vertices
  Initialize $trees \leftarrow |V|$
  ▷ Initialize the root of each vertex to itself
  Initialize $R \leftarrow v \forall v = v$
  Initialize $R2 \leftarrow v \forall v = v$
  while $trees > 1$ **do**
    ▷ Assign max value as the min weight for each vertex
    Initialize $W \leftarrow v \forall v = numericLimit$
    ▷ Assign max value as the min neighbor for each vertex
    Initialize $N \leftarrow v \forall v = numericLimit$
    ▷ Execute filter operator on all edges in F in parallel to find the minimum weight of the outgoing edges for each tree
    for $e \in F$ **do**
      ▷ If the edge is not a minimum for its tree
      if $getMinWeights(e, W, R) == false$ **then**
        ▷ Remove e from the frontier
        $F = F - \{e\}$
      end if
    end for
    ▷ Execute parallel for operator on all edges in F to find the minimum weight neighbor for each tree
    for $e \in F$ **do**
      $getMinNeighbors(e, W, N, R)$
    end for
    ▷ Execute parallel for operator on all vertices in V to add minimum outgoing edges for each tree to the MST
    for $v \in V$ **do**
      $addToMST(v, W, N, R, R2)$
    end for
    ▷ Execute parallel for operator on all vertices in V to get the updated root for each vertex
    for $v \in V$ **do**
      $updateRoots(v, R2)$
    end for
    ▷ Update roots to new roots
    $R = R2$
  end while
end procedure

Algorithm 3 MST getMinWeights Function

Input: Edge $e = (source, dest, weight)$, Array W, Array R
Output: Boolean true if the weight is a minimum, false otherwise

procedure GETMINWEIGHTS($\mathbf{e}, \mathbf{W}, \mathbf{R}$)
    ▷ If $source < dest$ and $source$ and $dest$ are not part of the same tree
    if $source < dest$
      & $R[source] \neq R[dest]$ **then**
        ▷ Use atomic min to update the min weight for both roots
        $oldWeightSource = atomicMin(W[R[source]], weight)$
        $oldWeightDest = atomicMin(W[R[dest]], weight)$
        ▷ If this is the min for either root, return true to keep the edge in the frontier
        if $weight <= oldWeightSource$
          or $weight <= oldWeightDest$ **then**
            return $true$
        end if
        ▷ Return false to remove the edge
        return $false$
    end if
end procedure

Algorithm 4 MST getMinNeighbors Function

Input: Edge $e = (source, dest, weight)$, Array W, Array N, Array R

procedure GETMINNEIGHBORS($\mathbf{e}, \mathbf{W}, \mathbf{N}, \mathbf{R}$)
    ▷ Check if this edge has the min weight for the source's root
    if $weight == W[R[source]]$ **then**
        ▷ Use atomic min to update the min neighbor to the min edge index with the min weight
        $atomicMin(N[R[source]], e)$
    end if
    ▷ Check if this edge has the min weight for the dest's root
    if $weight == W[R[dest]]$ **then**
        ▷ Use atomic min to update the min neighbor to the min edge index with the min weight
        $atomicMin(N[R[dest]], e)$
    end if
end procedure

will never attempt to connect to each other with two different edges.

The *addToMST* function shown in Algorithm 5 finally updates our MST. It operates on all vertices in the graph and first checks that the vertex is a root. Then, it gets the edge with the minimum neighbor of this root. If this vertex is not the root of the edge's source, it flips the source and the destination. We then know that the destination of this edge is not yet connected to our current vertex / root.

Then, the function checks that either the source is less than the destination or that the destination's root has a different minimum-neighbor edge. In the *getMinNeighbors* function, we ensured that no two neighboring trees would attempt to connect to each other with different edges. However, they could still attempt to connect to each other with the same edge. In that case, the source and destination would have been flipped above in one instance but not the other. Thus *source* $<$ *dest* will also fail for one and not the other. This check ensures that no duplicate edges are added. If the check passes, we finally add the edge to the MST! The MST weight is updated atomically, the number of trees is decremented atomically, and the root of this vertex is atomically updated to the root of the destination. In our check above to ensure the destination's root has a different minimum-neighbor edge, we need the value of the destination's root *prior to any updates that occur in this function*. Thus we maintain a separate vector for our new roots: $R2$. The function uses the old root values from R in its conditional checks, but updates values in $R2$.

As in the sequential implementation, when one tree is joined to another and the root is updated, the other vertices in the tree would not have their roots updated. Thus we must jump from root to root until we find the updated root for each vertex. Function *updateRoots*, shown in Algorithm 6, visits each vertex and stores the vertex's root as u. It then continually updates u to equal its own root until the root of u equals u. Then it sets the root of the vertex to u. This function looks exactly the same as the sequential version (which was omitted for simplicity) except that it is called later and in parallel.

Algorithm 5 MST addToMST Function

Input: Vertex v, Array W, Array N, Array R, Array $R2$, trees

```
1:  procedure ADDTOMST(v, W, N, R, R2, trees)
2:      ▷ If this vertex is a root
3:      if W[v] ≠ numericLimit then
4:          ▷ Get the edge to the root's min weight neighbor
5:          e(source, dest, weight) = N[v]
6:          ▷ Swap the source and destination if the root of the source is not v
7:          if R[source] ≠ v then
8:              temp = source
9:              source = dest
10:             dest = temp
11:         end if
12:         ▷ If the source is less than the dest or the reverse edge is not included
13:         if source < dest or N[R[dest]] ≠ e then
14:             ▷ Atomically add weight to the MST weight
15:             atomicAdd(w, weight)
16:             ▷ Atomically decrement the number of trees
17:             atomicAdd(trees, −1)
18:             ▷ Atomically set the (new) root of the source to the (new) root of the dest
19:             atomicExch(R2[v], R2[dest])
20:         end if
21:     end if
22: end procedure
```

The *updateRoots* function uses the root values in $R2$ which have the updates from *addToMST*. After *updateRoots* completes, we copy $R2$ to R.

We do not call to update the roots before adding edges to the MST as we did in the sequential algorithm. In the sequential version, updating the roots allowed us to ensure that two trees had not already been connected by another edge within the same **for** loop. We cannot use the same check in the GPU program as roots could be updated between our call to *updateRoots* and our check of the root value. Instead, we prevent loops by ensuring that no two trees can attempt to connect to one another with two different edges in *getMinNeighbors* and prevent duplicates in *addToMST*.

While the sequential version has two **for** loops, our parallel implementation has four parallel **for**s. In the sequential implementation, the minimum weight and neighbor could be found within the same loop. In the parallel implementation, we must find the minimum-

Algorithm 6 MST updateRoots Function

```
Input: Vertex v, Array R2
 1: procedure UPDATEROOTS(v, R2)
 2:     ▷ Set u to the (new) root of v
 3:     u = R2[v]
 4:     ▷ While the root of u is not u
 5:     while R2[u] ≠ u do
 6:         ▷ Set u to its root
 7:         u = R2[u]
 8:     end while
 9:     ▷ Set the root of v to u
10:     R2[v] = u
11: end procedure
```

weight edge with the minimum index to prevent loops as described above. Thus we find the minimum neighbor after the minimum weight has been found and two separate parallel **for**s are needed. *updateRoots* gets its own parallel **for** as well since the roots cannot be updated within *addToMST* in the parallel version.

On the GPU, intuition can be deceiving. Combining *getMinWeights* and *getMinNeighbors* certainly sounds more efficient code-wise. However, the minimum weight and minimum neighbor cannot be updated simultaneously and preventing this would require a critical section and lock. Pseudocode for this implementation is shown in Algorithm 7. However, this implementation has flaws. The practice of using critical sections with locks is widely-known to poorly utilize GPU resources. Moreover, if a branch occurs between when the lock is acquired and released in the machine code, thread divergence can lead to deadlock. The CUDA compiler can also schedule instructions in unexpected ways. Even if the implementation works on one CUDA architecture, it may not on another. For these reasons, the critical section with lock practice is highly discouraged in CUDA. On the GPU, the expectation and reality often do not align. GPU programmers must adapt to the unique constraints of the GPU. Best practices in single-threaded programming can hurt performance, or be infeasible altogether, on the GPU.

Algorithm 7 MST getMinWeights + getMinNeighbors with Lock

▷ Initialize the lock on each vertex to 0
Initialize $L \leftarrow 0 \forall v = v$
for $e \in F$ **do**
    $getMinWeightsMinNeighbors(e, W, N, R)$
end for

Input: Edge $e = (source, dest, weight)$, Array W, Array N, Array R
procedure GETMINWEIGHTSMINNEIGHBORS($\mathbf{e}, \mathbf{W}, \mathbf{N}, \mathbf{R}$)
    ▷ Initialize our loop conditional variable
    $waiting = true$
    while waiting **do**
        ▷ If the weight is less than or equal to the min
        if $weight \leq W[R[source]]$ **then**
            ▷ If the lock is acquired
            if $AtomicCAS(L[R[source]], 0, 1) == 0$ **then**
                ▷ Check again if the weight is min (it may have changed)
                ▷ If it is a new min, set min weight and neighbor
                if $weight < W[R[source]]$ **then**
                    ▷ Atomic operations or a memory fence are needed to ensure other threads see updates
                    $AtomicExch(W[R[source]], weight)$
                    $AtomicExch(N[R[source]], e)$
                else if $weight == W[R[source]]$ **then**
                    ▷ If it equals the current min, check if neighbor is min
                    $AtomicExch(N[R[source]], \min(e, N[R[source]]))$
                end if
                ▷ Release the lock
                $AtomicExch(L[R[source]], 0)$
                ▷ Exit the while loop
                $waiting = false$
            end if
        else
            ▷ Exit the while loop if this is not a min
            $waiting = false$
        end if
    end while
end procedure

Datset	Vertices	Edges
ga2010	291,086	1,418,056
pdb1HYS	36,417	4,344,765
Ga41As41H72	268,096	18,488,476
CurlCurl_4	2,380,515	26,515,867
af_shell10	1,508,065	52,672,325
GAP-road	23,947,347	57,708,624
audikw_1	943,695	77,651,847
dielFilterV3real	1,102,824	89,306,020
nlpkkt120	3,542,400	96,845,792
Flan_1565	1,564,794	117,406,044

Table 4.1: Datasets selected to test MST

4.4 Performance Analysis

We use our instrumentation code to conduct performance analysis on our parallel MST algorithm. MST requires a symmetric, weighted graph with one connected component. Thus we select 10 graphs of various sizes from the SuiteSparse Matrix Collection [1] that meet our requirements. These datasets are shown in Table 4.1.

We graph MTEPS, runtime, and edges visited against the number of edges to see how our algorithm performs on datasets of different sizes. Note that our algorithm itself has linear complexity.

Figure 4.1 shows MTEPS of our MST algorithm vs the number of edges in the graph. When the graph size increases, we would expect gains from additional parallelism and thus higher MTEPS. As expected, MTEPS increases as the size of the graph increases. Nothing here points to an issue in our algorithm. If the MTEPS pattern were to appear

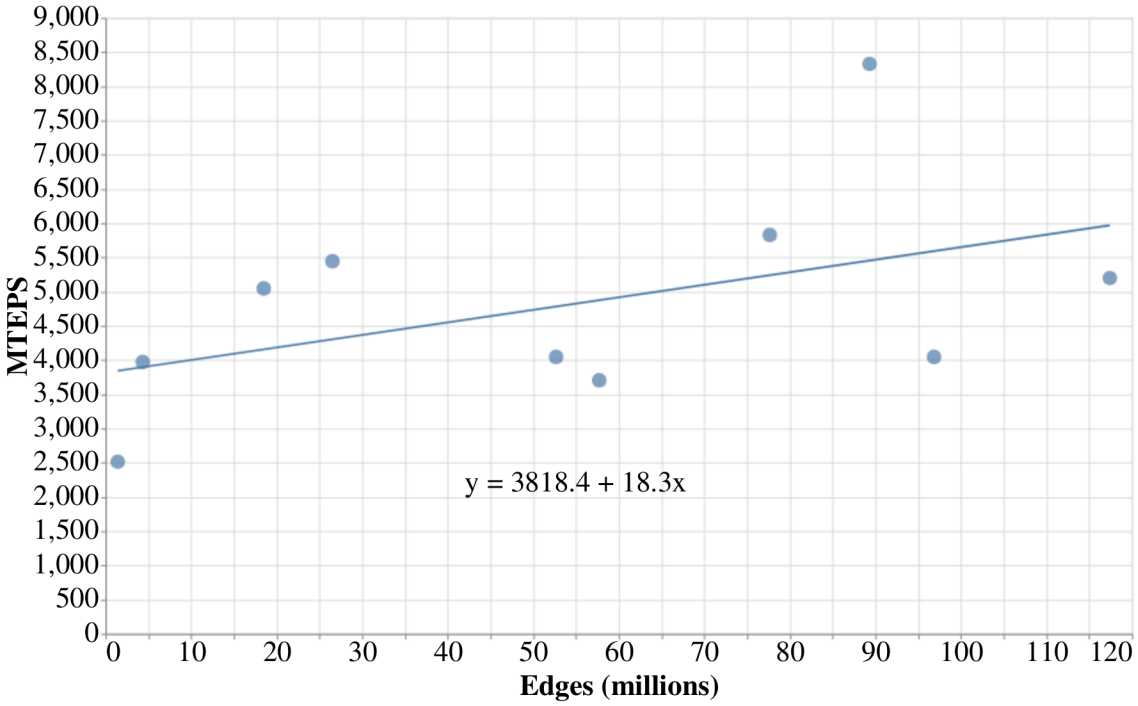

Figure 4.1: MST MTEPS vs number of edges; Tesla V100 GPU; various datasets

more logarithmic, that would signal that the MTEPS were not scaling well as the graph size increased. Figure 4.2 plots the MST runtime as a function of the number of edges in the graph. This also follows the trend we would expect; as the number of edges increases, so does the runtime. If this plot showed a more exponential pattern, it would point to a scaling problem with our algorithm.

Figure 4.3 plots the number of edges visited by our MST algorithm against the number of edges in the graph. Based on the slope of the line, we see that for each additional edge in the graph, we visit an average of 7.2 more edges in our algorithm. This may be something we can improve upon through better frontier management.

In figure 4.4, we plot MTEPS vs average degree. The degree of a vertex is its number of edges. The datasets we chose are nearly all from different applications, but similar types of datasets will often have similar average degrees. According to the plot, there is

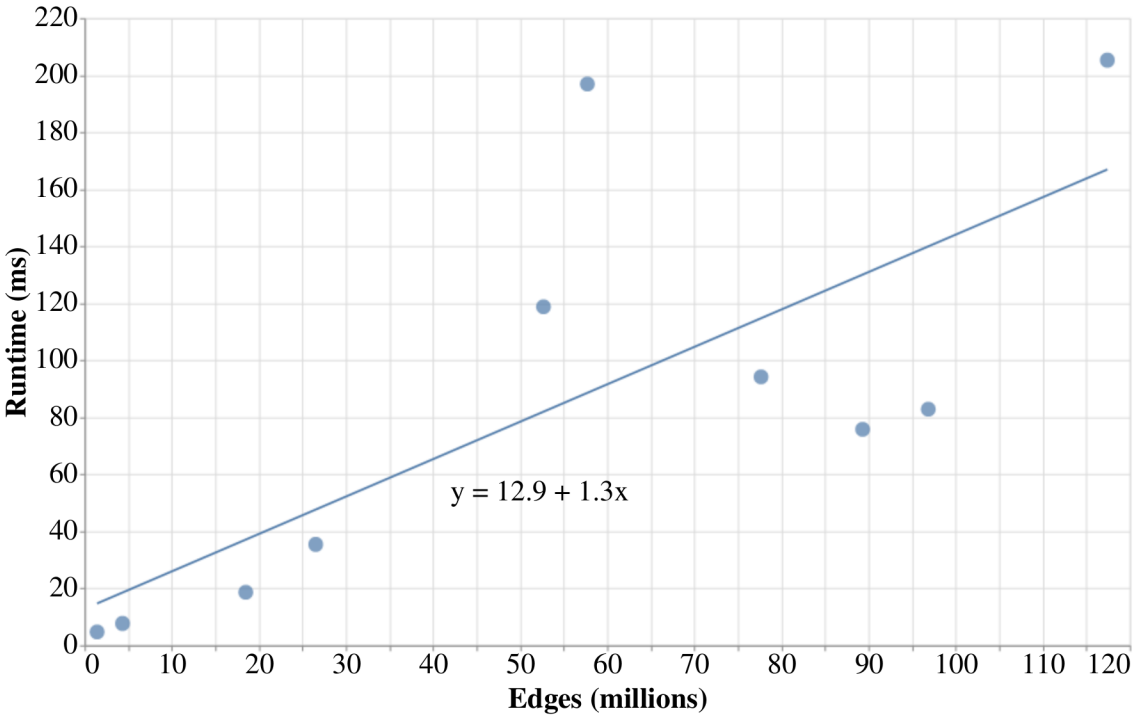

Figure 4.2: MST runtime vs number of edges; Tesla V100 GPU; various datasets

no apparent relationship between the two variables. Looking at our algorithm, we would not expect a strong relationship. In the first two parallel **for**s, we iterate over all edges in the graph. Then, we visit all vertices twice. There are no **for** loops nested within these parallel **for**s. Some of our other algorithms, such as Graph Coloring, include a nested **for** loop that iterates over all neighbors of a vertex. We would expect a stronger relationship in such cases. For our MST algorithm, however, the lack of relationship here is not alarming.

4.5 Optimizations

Taking in our observations from the previous section, we look for ways to optimize our MST algorithm. The slope in the edges visited vs edges plot tells us that for each additional edge in our graph, we visit 7.2 additional edges on average. Are all these edge accesses

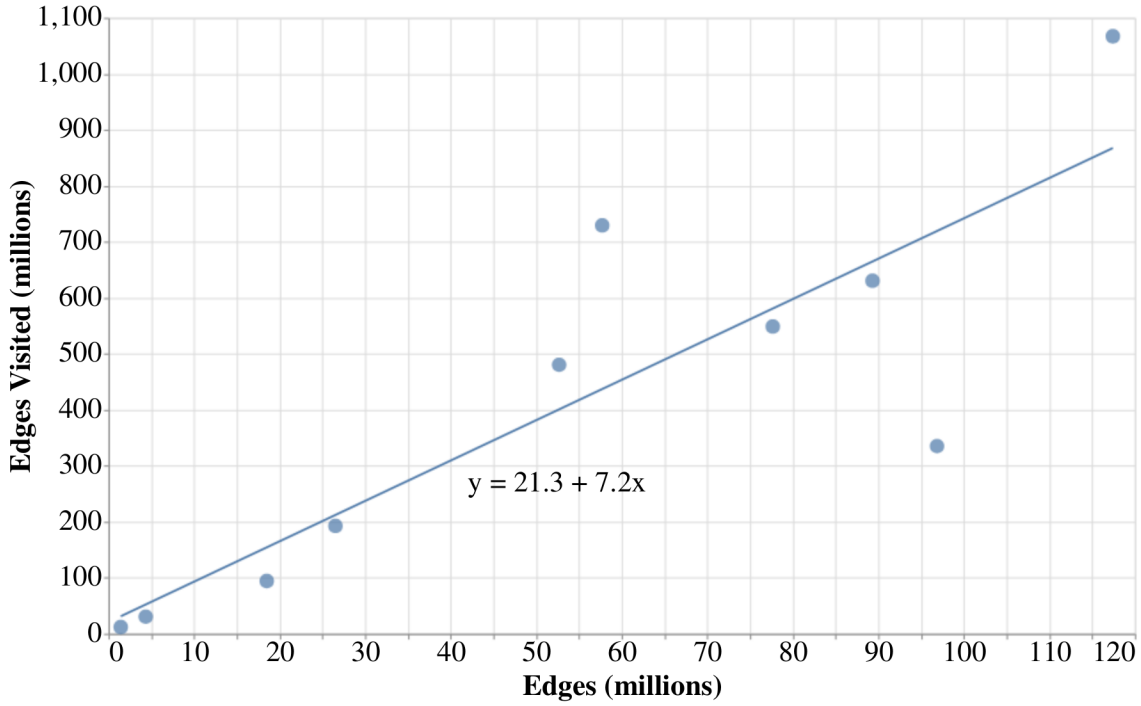

Figure 4.3: MST edges visited vs number of edges; Tesla V100 GPU; various datasets

truly necessary? This question sheds light on the fact that our algorithm does very little in terms of frontier management. Two of our parallel **for**s visit each edge in every iteration! Better frontier management could help us cut out superfluous edge visits.

Looking over our algorithm through this lens, we notice that in every iteration, *getMinWeights* filters out edges where the source is greater than the destination. The purpose of this is to operate on an edge and its reverse together to improve performance. However, this condition will filter out the same edges in every iteration. There is no reason to repeat the same check over and over again!

To remedy this, we add a function with the sole purpose of filtering out these edges, and only call it once in the first iteration. We maintain the frontier outputted by this function to use as the input frontier for *getMinWeights* in every iteration. This function, *getMatrixHalf*, effectively filters out half of the matrix. The half we keep is the upper right

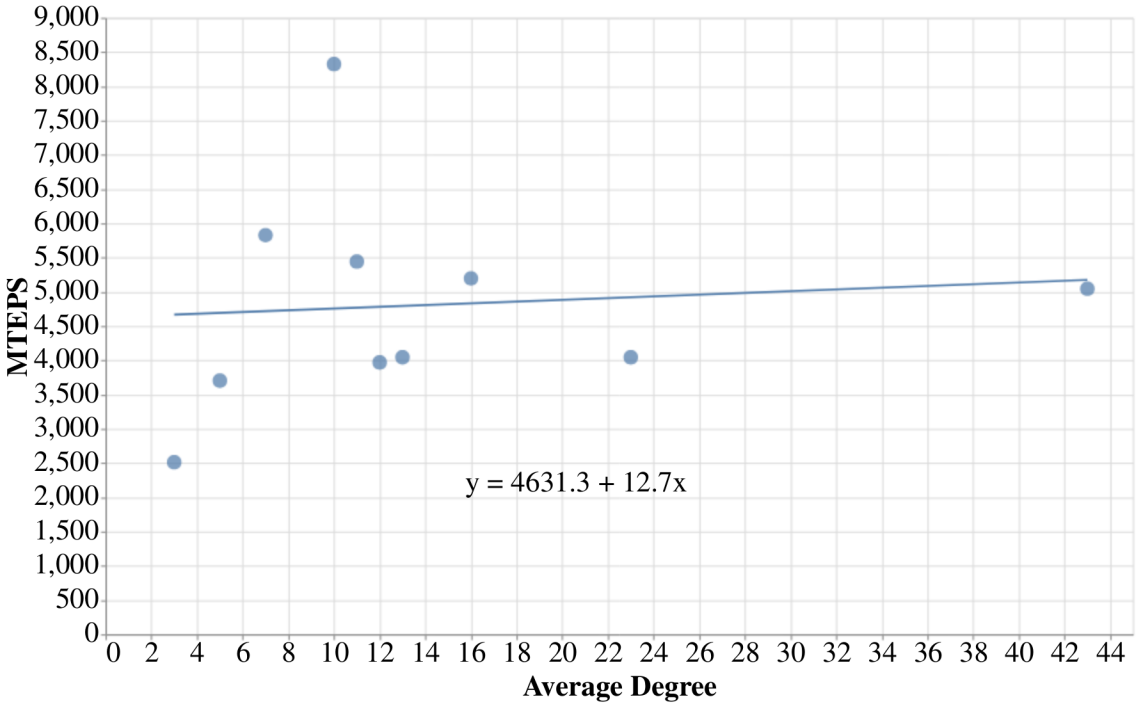

Figure 4.4: MST MTEPS vs average degree; Tesla V100 GPU; various datasets

half. The function is shown in Algorithm 8.

Algorithm 8 MST getMatrixHalf Function

Input: Edge $e = (source, dest, weight)$
Output: Boolean true if the source is less than the destination, false otherwise

```
1: procedure GETMATRIXHALF(e)
2:     ▷ If the source is less than the destination
3:     if source < dest then
4:         ▷ Return true to keep the edge in the frontier
5:         return true
6:     end if
7:     ▷ Return false to remove the edge
8:     return false
9: end procedure
```

We modify our overall algorithm to incorporate this function. Like *getMinWeights*, it is called by our filter operator, which operates on a frontier in parallel and filters out all elements for which the function evaluates to false. Algorithm 9 shows these updates. We

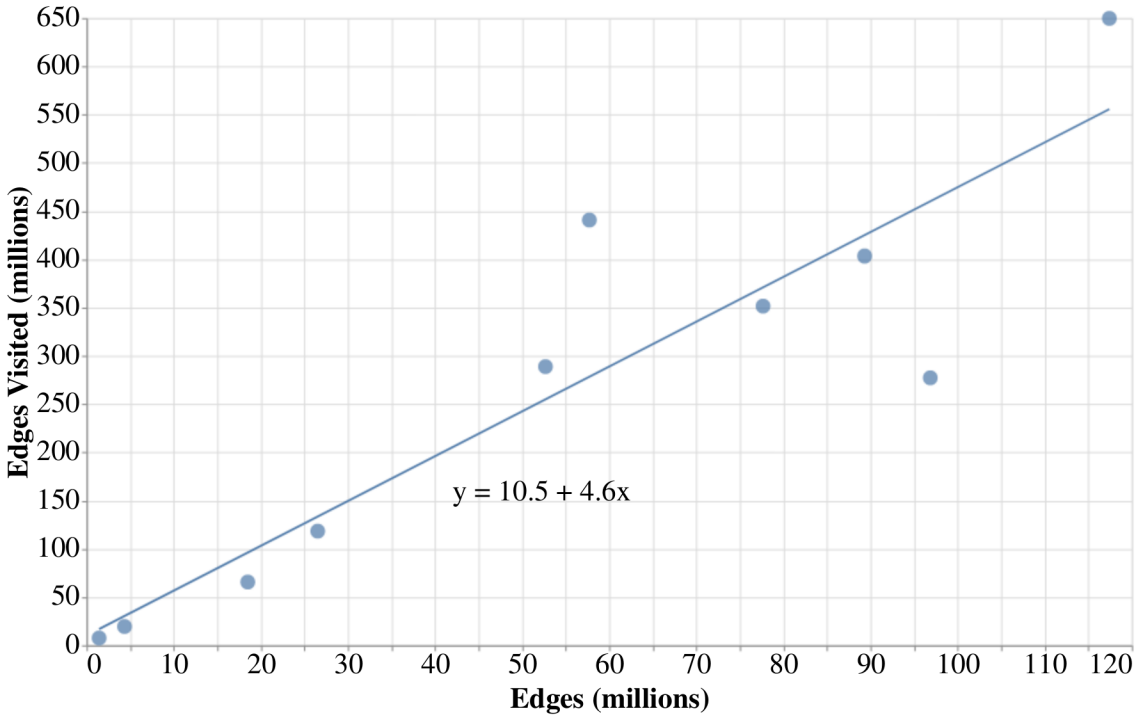

Figure 4.5: MST edges visited vs edges (optimized algorithm); Tesla V100 GPU; various datasets

also update *getMinWeights* to no longer check that the source is less than the destination.

4.6 Comparison

We now compare our original and optimized implementations.

Figure 4.5 plots edges visited against edges for our optimized algorithm. While the slope of the line for our original implementation was 7.2, the figure shows that our new slope is 4.6. That means we are visiting an extra 4.6 edges per additional edge in the graph instead of 7.2! Figure 4.6 shows the reduction in edges visited for each dataset.

To compare the general performance of our implementations, we look at speedup and a comparison of MTEPS. Figure 4.7 shows the speedup from our optimizations for each dataset. In all cases, the runtime is lower for our optimized implementation. Our opti-

Algorithm 9 Gunrock Optimized Parallel MST Algorithm

Input: Graph G=(V,E)
Output: Float w MST weight

```
1:  procedure GUNROCKMST(G)
2:      ...
3:      ...
4:      ...
5:      while trees > 1 do
6:          ...
7:          ...
8:          ...
9:          ▷ If this is the first iteration
10:         if iteration == 0 then
11:             ▷ Execute filter operator on all edges in F in parallel to filter out edges with source greater than destination
12:             for e ∈ F do
13:                 ▷ If the source is greater than the destination
14:                 if getMatrixHalf(e) == false then
15:                     ▷ Remove e from the frontier
16:                     F = F − {e}
17:                 end if
18:             end for
19:         end if
20:         ▷ Do not modify F further as we need it in future iterations
21:         F2 = F
22:         ▷ Execute filter operator on all edges in F2 in parallel to find the minimum weight of the outgoing edges for each tree
23:         for e ∈ F2 do
24:             ▷ If the edge is not a minimum for its tree
25:             if getMinWeights(e, W, R) == false then
26:                 ▷ Remove e from the frontier
27:                 F2 = F2 − {e}
28:             end if
29:         end for
30:         ▷ Execute parallel for operator on all edges in F2 to find the minimum weight neighbor for each tree
31:         for e ∈ F2 do
32:             getMinNeighbors(e, W, N, R)
33:         end for
34:         ...
35:         ...
36:         ...
37:     end while
38: end procedure
```

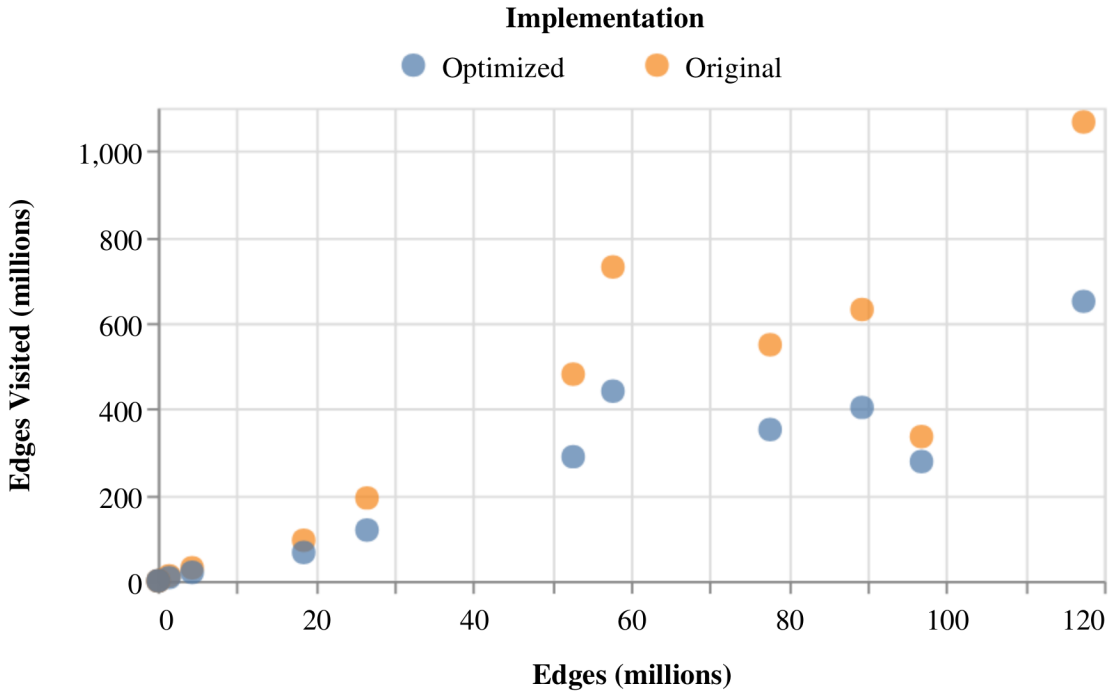

Figure 4.6: MST edges visited vs edges by implementation; Tesla V100 GPU; various datasets

mizations decrease runtime by up to 22%, with a 17% reduction being the average.

Figure 4.8, which shows MTEPS vs edges for both implementations across our datasets, tells another story. MTEPS is *lower* for our optimized algorithm than for our original. In light of type of optimizations we made, this also makes sense. In the original algorithm, *getMinWeights* visits each edge in the graph and immediately returns false if the destination is less than the source. Moreover, it does this in each iteration. Those are cheap edge visits that our optimized algorithm eliminates. Most of the edge visits that are left are more expensive. Thus it is no surprise that the optimized algorithm comes with lower MTEPS.

Note that the second largest dataset (nlpkkt120) has a search depth of only three; thus it does not undergo enough iterations to have a large reduction in edges visited or MTEPS. Though close, the MTEPS of the optimized algorithm is still lower than that of the original.

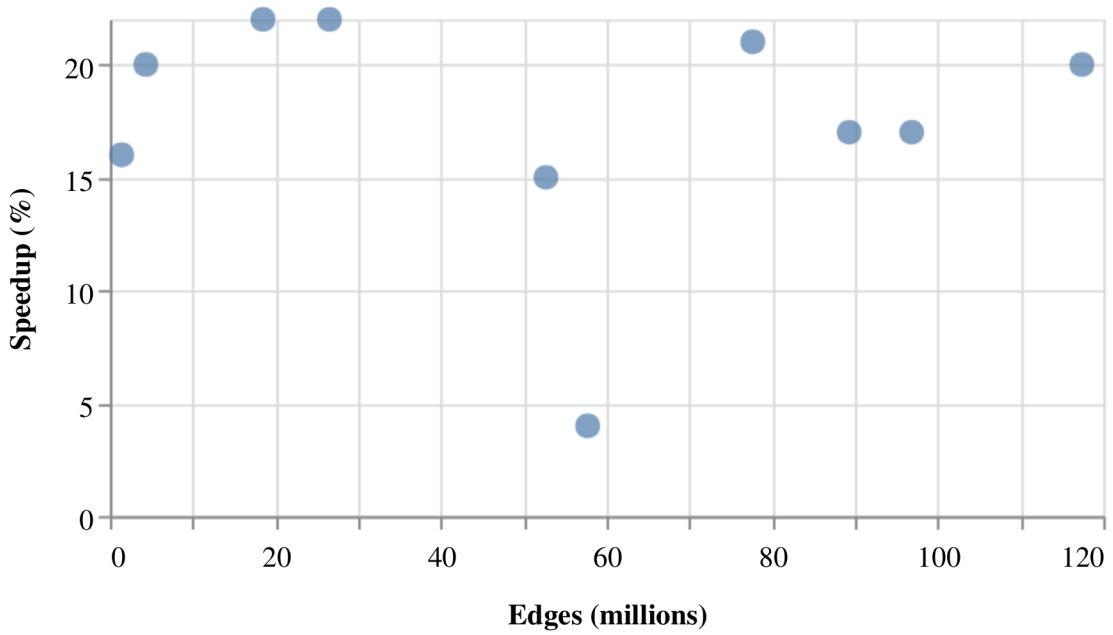

Figure 4.7: MST speedup vs edges; Tesla V100 GPU; various datasets

The phenomenon we see here is not unique to graph algorithms, or even to the GPU. A performance analysis metric that is similar to but more general than MTEPS is millions of instructions per second (MIPS). Like MTEPS, it gives a measure of how much total work a program is doing. When programs are compiled with an optimizing compiler, MIPS will often decline; the extra instructions that the compiler removes are often simple instructions that artificially elevate MIPS.

This shows us that MTEPS, and similar metrics, cannot be looked at in isolation. They are useful for seeing how much total work an algorithm does over a unit of time, however that does not tell the full story. We could certainly throw in a bunch of cheap, superfluous edge visits and improve our MTEPS. That is essentially what the original algorithm unintentionally does in fact. Our optimized algorithm does the same job in less time; thus it is undeniably more efficient, despite having lower MTEPS.

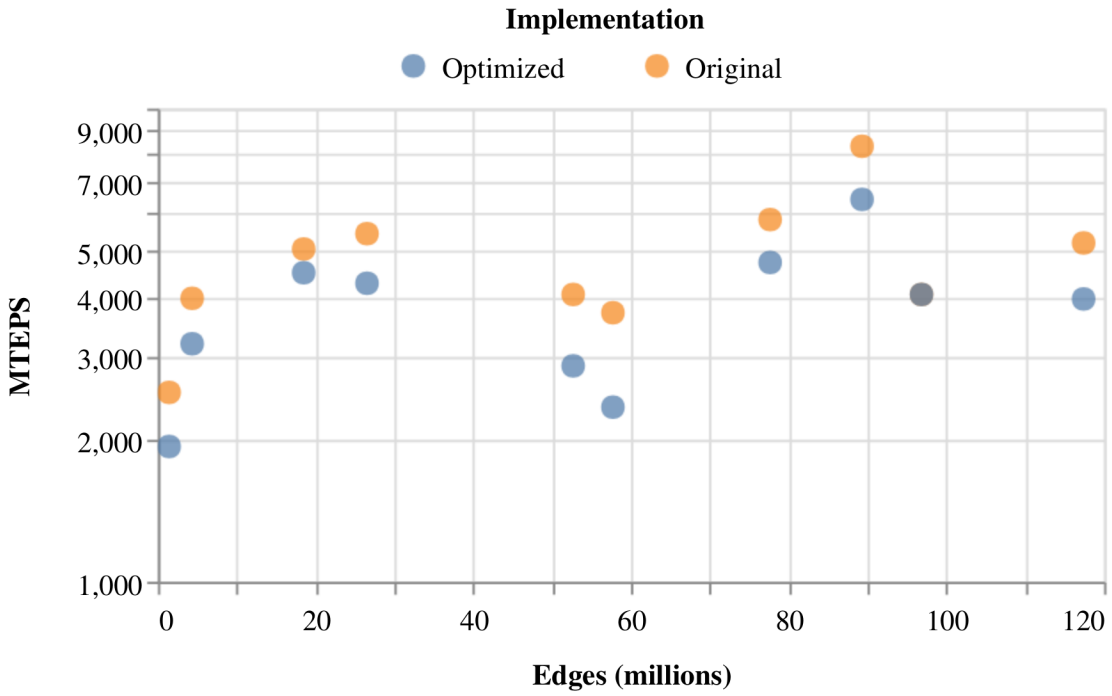

Figure 4.8: MST MTEPS vs edges by implementation; Tesla V100 GPU; various datasets

Chapter 5

Conclusion

Optimizations based on insights from performance data can speed up GPU graph algorithms. In this paper we presented our design and implementation of performance analysis in Gunrock 2.0. We discussed competing design goals and how to adapt the implementation based on different goals. We also demonstrated how to conduct performance analysis and optimizations with a case study using our MST algorithm.

When performance can be measured, it can be improved. Employing the techniques discussed in this paper will allow developers to provide strong performance for the applications that need it most, from gaming to national security.

Chapter 6

Summary of Contributions

All the work mentioned in this paper was completed by myself, Annie Robison, with guidance from my advisor Professor John Owens, and my mentor, Muhammad Osama. All of my code contributions to the Gunrock 2.0 framework are summarized in Table 6.1. I first developed our minimum spanning tree algorithm, which I later optimized while writing this paper. I next added NVBench benchmarking support for all of our algorithms. This effort required communication with NVIDIA, as our programs have a different structure than NVBench expects. I then added performance analysis support as described in this paper. Finally, I updated our graph builder to support graph properties and multiple simultaneous storage formats. The graph builder changes will allow algorithm developers to use graph properties without having to extract them themselves, and write algorithms that require both forward and backward traversing of a graph, such as direction-optimized BFS.

Contribution	Lines of code added	Lines of code removed
Minimum spanning tree algorithm	+994	-508
NVBench benchmarking support	+2733	-1001
Performance analysis support	+4561	-3194
Graph builder memory and design rework	+2043	-1445
Total	+10331	-6148

Table 6.1: Code Contributions to Gunrock 2.0

REFERENCES

[1] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[2] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/169411.

[3] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1):3:1–3:49, August 2017. doi: 10.1145/3108140. URL http://escholarship.org/uc/item/9gj6r1dj.

[4] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149, 2014. doi: 10.1109/IISWC.2014.6983053.