An Automated Framework for Concurrent Graph Processing on GPU

by

Mandana BagheriMarzijarani

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

25 December 2023

An Automated Framework for Concurrent Graph Processing on GPU

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Mandana BagheriMarzijarani

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Didem Unat (Advisor)

_____

Prof. Dr. Öznur Özkasap

_____

Assoc. Prof. Kamer Kaya

Date: _____

ABSTRACT

An Automated Framework for Concurrent Graph Processing on GPU
Mandana BagheriMarzijarani
Master of Science in Computer Science and Engineering
25 December 2023

GPUs have become the bleeding edge in high-performance systems used for artificial intelligence and scientific computations. Running parallel jobs and algorithms on GPUs yields results that are orders of magnitude faster than running the same jobs on CPUs that employ fewer cores.

A significant body of computational tasks incorporates graph traversals and computations associated with traversal of large graphs. These tasks are either graph traversal tasks in nature or are simplified or transformed into graph traversal tasks.

Kernel fusion has been proposed in several bodies of research as a means of fusing computation cores (kernels) in parallel jobs so that the performance can be optimized. Specifically, kernel fusion is used to fuse accesses to same graph nodes and corresponding regions of memory by multiple parallel jobs so that memory caches are more efficiently utilized, resulting in performance improvements. As kernel fusion's fundamental idea is sharing resource access to increase utilization by multiple jobs, it is even better suited for GPUs that run hundreds of jobs in parallel than CPUs that run a few jobs in parallel. However, kernel fusion is a computationally expensive operation that incorporates control flow changes and manipulation of the order of execution for computational jobs. As such, kernel fusion is harder to utilize on GPUs compared to CPUs.

This work introduces a streamlined kernel fusion framework for concurrent graph processing on GPUs. The framework enables definition and implementation of graph jobs that can be executed in parallel in GPUs, and are automatically fused by the kernel fusion algorithm implemented in the framework, to achieve better performance. The framework also introduces novel data handling structures, as well as a meta-compiler that enables static polymorphism for running multiple different jobs as a job queue on the GPU, without performance hits that are associated with typical polymorphism.

The framework is then extensively evaluated by defining four common graph jobs (BFS, SSSP, PageRank, Label Propagation) and running up to 200 parallel instances of these jobs in homogeneous and heterogeneous manners, with and without kernel fusion. The evaluations show that kernel fusion provides about 5% to 10% performance improvement when job parallelism is reasonably high (i.e., more than 10 parallel jobs).

ÖZETÇE

Eşzamanlı Grafik İşleme için Otomatik GPU Çerçevesi
Mandana BagheriMarzijarani
Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans
25 Aralık 2023

GPU'lar, yapay zeka ve bilimsel hesaplamalar için kullanılan yüksek performanslı sistemlerde son teknoloji haline geldi. GPU'larda paralel işler ve algoritmalar çalıştırmak, aynı işleri daha az çekirdek kullanan CPU'larda çalıştırmaktan çok daha hızlı sonuçlar verir.

Hesaplamalı görevlerin önemli bir kısmı, grafik geçişlerini ve büyük grafiklerin geçişiyle ilişkili hesaplamaları içerir. Bu görevler ya doğası gereği grafik geçiş görevleridir ya da basitleştirilmiş veya grafik geçiş görevlerine dönüştürülmüştür.

Çekirdek füzyonu, performanslarının optimize edilebilmesi için hesaplama çekirdeklerin paralel işlerde birleştirilmesinin bir yolu olarak çeşitli araştırma gruplarında önerilmiştir. Spesifik olarak çekirdek füzyonu, aynı grafik düğümlerine ve ilgili bellek bölgelerine birden fazla paralel iş tarafından yapılan erişimleri birleştirmek için kullanılır, böylece bellek önbellekleri daha verimli kullanılır ve bu da performans iyileştirmesiyle sonuçlanır. Çekirdek füzyonunun temel fikri, birden fazla işin kullanımını artırmak için kaynak erişimini paylaşmak olduğundan, yüzlerce işi paralel olarak çalıştıran GPU'lar için, birkaç işi paralel olarak çalıştıran CPU'lardan bile daha uygundur. Bununla birlikte, çekirdek füzyonu, kontrol akışı değişikliklerini ve hesaplamalı işler için yürütme sırasının manipülasyonunu içeren, hesaplama açısından pahalı bir işlemdir. Bu nedenle, çekirdek füzyonunun GPU'larda kullanılması CPU'lara kıyasla daha zordur.

Bu çalışma, GPU'larda eşzamanlı grafik işleme için geliştirilmiş bir çekirdek füzyon çerçevesi sunmaktadır. Çerçeve, GPU'larda paralel olarak yürütülebilen ve daha iyi performans elde etmek için çerçevede uygulanan çekirdek füzyon algoritması tarafından otomatik olarak birleştirilen grafik işlerinin tanımlanmasına ve uygulanmasına olanak tanır. Çerçeve aynı zamanda yeni veri işleme yapılarının yanı sıra, tipik polimorfizmle ilişkili performans düşüşleri olmadan GPU'da bir iş kuyruğu olarak birden fazla farklı işi çalıştırmak için statik polimorfizm sağlayan bir meta derleyiciyi de içeriyor.

Bu tez, dört ortak grafik işi (BFS, SSSP, PageRank, Label Propagation) tanımlanarak ve bu işlerin 200'e kadar paralel örneğinin çekirdek füzyonu ile ve çekirdek füzyonu olmadan homojen ve heterojen şekillerde çalıştırılmasıyla kapsamlı bir şekilde çerçeveyi değerlendirir. Değerlendirmeler, iş paralelliği makul derecede yüksek olduğunda (yani 10'dan fazla paralel iş) çekirdek füzyonunun yaklaşık %5 ila %10 performans artışı sağladığını göstermektedir.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Didem Unat for her patience and invaluable guidance throughout my research journey, without her support I would not have made it through my masters degree.

I am grateful to Prof. Kamer Kaya who generously provided guidance and expertise during the course of this research.

I would like to also extend my appreciation to my defense committee member, Prof. Öznur Özkasap for her great feedback and guidance.

Additionally I acknowledge with appreciation the funding that this project has received in part by the European High-Performance Computing Joint Undertaking under grant agreement No 956213, by the Turkish Science and Technology Research Centre Grant No 120N003 and from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

Last but not least, I am grateful for the love, encouragement and support that my family, specially my husband and my son have given me to be able to undertake this endeavour.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SCC	Strongly Connected Component
DMA	Direct Memory Access
BFS	Breadth-First Search
SSSP	Single-Source Shortest Path
PR	PageRank
LP	Label Propagation
DAG	Directed Asyclic Graph

Chapter 1

INTRODUCTION

Kernel fusion refers to the optimization technique in computer programming where multiple computational operations, known as kernels, are combined or fused into a single, more efficient unit of execution. Kernel fusion can achieve this efficiency by reducing the overhead associated with preparing and launching sequential kernels, as well as by providing more cohesive access to resources such as memory, resulting in better cache utilization.

1.1 Overview

Kernel fusion can be done automatically or manually. Automated kernel fusion is a process where software tools or compilers automatically combine multiple computational kernels into a single optimized unit. In contrast, manual kernel fusion is done ad-hoc and by developers, via combining two or more kernels manually into an optimized unit.

Kernel fusion has been extensively performed on CPUs [Chen et al., 2021, Pan and Li, 2017, Mazloumi et al., 2019, Xue et al., 2014, Zhao et al., 2019] to achieve increases in performance and reduction in energy usage. However, GPUs are the leading accelerator in modern High Performance Computing (HPC) systems, equipping 7 of the 10 leading Top500 [Strohmaier, 2006] supercomputers in the world today. GPUs are frequently used to solve scientific and computational problems, as they offer hundreds to thousands of cores in contrast with dozens on a CPU, albeit slower and with more limitations. GPUs are naturally much more suited to kernel fusion, as the degree of parallel execution of kernels on them is significantly higher, simply due to the increased availability of cores.

Considering the additional number of cores in GPUs, kernel fusion has been applied extensively in GPUs [Wang et al., 2010, Wahib and Maruyama, 2014, Filipovič et al., 2015], mostly through manual methods. There is multiple research that covers automated kernel fusion in domains such as machine learning and image processing. To the best of our knowledge, automatic kernel fusion for concurrent graph processing on GPUs has not yet been covered in existing research.

In recent years, graphs with billions of vertices have been used to represent a diverse number of real-world problems such as scientific computations, social networks, machine learning and biology. Speeding up graph processing algorithms can improve the efficiency of these computations. As such, many graph processing algorithms are proposed and the area is well studied in recent years [Zhang et al., 2018b, Chen et al., 2021, Shun and Blelloch, 2013, Pan and Li, 2017, Wang et al., 2016, Xue et al., 2014]. This work introduces an automated framework for concurrent graph processing kernel fusion in GPUs. The framework streamlines definition and implementation of graph algorithms (henceforth called graph jobs). The framework allows execution of many different jobs in tandem on the GPU. Furthermore, the framework is bundled with an optimized kernel fusion implementation for GPUs. The framework automatically performs kernel fusion on the executing jobs to increase their performance. Additionally, the framework is coupled with a meta-compiler that enables static (compile-time) polymorphism, facilitating high-speed execution of various compiled jobs at runtime.

The framework is primarily focused on graph traversal and processing jobs that work on a single large graph. It is believed that a vast majority of computational tasks can be approached as jobs involving the traversal and processing of graphs. The kernel fusion introduced in this work focuses on fusing memory accesses, via executing jobs that access similar nodes in the graph input, resulting in more effective memory caching.

The framework is chiefly implemented in C++ with supporting scripts in other languages. Evaluations show that the framework is effective in increasing runtime performance of a large number of parallel jobs by about 5% to 10%, compared to

running the same jobs without kernel fusion.

1.2 Thesis Contributions

The following are contributions of this thesis:

- Proposing a graph fusion technique for concurrent graph processing on GPUs by using a subset of frontier of different graph jobs as a proxy for memory access.

- Development and implementation of our novel, *automated kernel fusion framework*. The framework allows the programmers to define their own graph jobs and automatically generates the fused version of multiple graph processing jobs. The framework also divides the execution of kernel fusion steps between CPU and GPU to achieve the optimal overhead, .

- Pioneering of an original meta-compiler to enable memory-efficient shared data structures for computational jobs and support of static polymorphism. The meta-compiler overcomes the challenges of native polymorphism and Runtime Type-Information (RTTI) based polymorphism when being used partially on both the CPU and the GPU.

- Implementation of four graph processing jobs using the novel framework, namely Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR) and Label Propagation (LP).

- Evaluation of the framework with the implemented jobs grouped into several job sets that resemble the real-world graph processing workload both in complexity and number of parallel jobs on two platforms with combinations of up to 200 parallel jobs in each job set, demonstrating approximately 5% to 10% increase in runtime performance when enabling kernel fusion.

1.3 Thesis Organization

The rest of this document is organized as follows. Chapter 2 provides background and motivation for kernel fusion, automated kernel fusion, as well as kernel fusion in GPUs, and finally the scope of this work. Chapter 3 covers related works and existing body of research in kernel fusion, for both CPUs and GPUs. Chapter 4 covers the automated kernel fusion framework, its development challenges and novel ideas, including the architecture of the framework as well as the meta-compiler, and the kernel fusion algorithm. Evaluations are performed in Chapter 5. Chapter 6 provides a conclusion and covers potential future works.

Chapter 2

BACKGROUND & MOTIVATION

Kernel fusion as a research area has motivations rooted in the pursuit of optimizing computational efficiency and advancing parallel processing capabilities. The imperative to address the growing demand for high-performance computing solutions across diverse fields is the motivation for kernel fusion. As applications become increasingly complex and *data-intensive*, the need for innovative approaches to harness the full potential of CPUs and GPUs becomes paramount.

One key motivation for delving into kernel fusion is the desire to overcome the limitations posed by sequential execution of independent computational tasks and jobs. By consolidating and fusing kernels, research aims to exploit the inherent parallelism within these tasks on different architectures, thereby unlocking a pathway to significantly enhance the throughput and overall computational power. This consolidation not only translates into gains in performance but also holds the promise of accelerating a wide range of applications, from scientific simulations to machine learning algorithms.

2.1 GPU vs. CPU

While kernel fusion has found success across various domains within the realm of CPUs, there is currently not as many existing research addressing multi-algorithm automated kernel fusion on GPUs, to the best of our knowledge. The absence of progress in this regard raises suspicions for several reasons. The chief among them is that despite the success in pushing of the data path to the GPU, overall control of execution, i.e., the *control-flow graph* has stayed resiliently on the CPU. This is mostly because in GPU applications, the CPU serves as a barrier that synchronizes kernels with each other. Such synchronization is necessary in iterative applications,

and even more needed when kernel fusion is involved, as control-flow graphs need to be substantially modified to account for enabling and disabling specific kernels and their access to specific resources.

Furthermore, there are many engineering challenges for implementing automated GPU kernel fusion. How much of the logic should be implemented within the GPU, versus what part of it needs to remain in the CPU? Ideally, moving as much of the logic into the GPU as possible would result in better performance, less context switch and less memory overhead. Yet, the GPU has certain limitations, both in data and control paths, that pose significant challenges. For example, memory limits in the GPU such as limited and rigid heap size, less efficient heap allocation mechanisms, and lack of native memory paging (i.e., all memory needed by jobs need to be allocated and copied before execution), imply that only a limited number of jobs can be executed in parallel within the GPU without CPU meddling. On the other front, control-flow limits on the GPU include limited polymorphic support (e.g., virtual functions table for C++ needs to be entirely on the GPU to function properly, and is significantly slower than the CPU counterpart), as well as global synchronization and barrier challenges, such as when jobs need to be in synchronized after each iteration before kernel fusion can inspect their frontiers and fuse them for the next iteration.

Conversely, kernel fusion appears to be better suited for GPU workloads, provided that the associated engineering challenges can be effectively addressed. This alignment is particularly pronounced given the higher degree of parallelism inherent in GPUs. Overcoming engineering hurdles could lead to substantial gains in both performance and energy efficiency, especially in the case of the many-thousand-core GPUs widely utilized for computational and scientific tasks today, potentially yielding significant advantages.

Preliminary estimates indicate that achieving complete memory access cohesion on GPUs during the execution of 100 parallel jobs could result in a remarkable performance gain of up to 500x. These estimates were obtained by running identical breadth-first search graph traversal algorithms on a large graph, concurrently, 100 times. Two controlled environments were employed. One initiating the traversal

from the same node for all jobs, and the other starting the traversal from random nodes. The first experiment exhibited a speedup of 350 to 500 times compared to the second, primarily attributed to the efficient caching of memory accesses among all jobs.

This noteworthy speedup underscores the immense potential of GPU kernel fusion in enhancing runtime performance and curtailing the energy footprint of extensive computational parallel tasks.

2.2 Scope

To make this research achievable, the scope was limited in multiple dimensions. First and foremost, this research is limited to automation of concurrent graph processing kernel fusion on GPU, as it is believed that the grand majority of scientific computational tasks can be morphed or simplified to a graph processing task.

Secondly, all the jobs that are running in parallel will read from a single graph. Jobs that are independently working on multiple graphs are out of the scope for this work.

Thirdly, this work focuses on graph algorithms that can be described using the pull and push engines. In a pull engine, the computation is initiated by individual nodes, each of which actively pulls the necessary data from its neighboring vertices. In a push engine, each node pushes updates or computations to its neighboring vertices.

Fourth, the frontier of a job is defined as the next set of nodes that a job is going to access in the graph. The frontier generally starts by having a single node – the starting node, although this is job dependent and certain jobs can define it differently. The frontier then expands as the job iterations continue, usually reaching its zenith half-way across the iterations, and dwindling henceforth. This work assumes that the frontier of the graph is a proxy for memory regions that it is going to access. If a specific job for example, accesses nodes that are outside its frontier pervasively, it will not benefit from the kernel fusion designed in this work, due to negating its underlying assumption.

Fifth, the jobs need to run in a single GPU, as memory caching is only meaningful within one memory. Naturally, this limitation can be easily circumvented by grouping jobs per GPU, and fusing kernels for each device. However, for the sake of brevity the rest of this work assumes single GPU environments.

Sixth, this work focuses on achieving better performance and energy efficiency by increasing memory access cohesiveness of kernels, in contrast with other avenues of kernel fusion such as reduction of kernel launch overhead or control-flow streamlining.

Chapter 3

RELATED WORK

In this section, the current state of the art graph processing systems and their limitations are discussed. Subsequently, the proposed framework is differentiated from the existing works.

The current existing body of research can be categorized into four major divisions. Namely (1) CPU-Based single graph processing systems, (2) GPU-based single graph processing systems, (3) concurrent CPU-based graph processing systems and (4) concurrent GPU-based graph processing systems comprise these four categories. Table 3.1 summarizes the different graph processing prior research and the challenges they have each striven to resolve. Table 3.2 provides a brief comparison between our framework and another framework in prior research that was defined and evaluated on a different architecture.

3.1 CPU-based Graph Processing Systems

The body of research in CPU-based graph processing systems has aimed to offer optimized frameworks for development and processing of a *single* graph algorithm at a time. Although this single graph algorithm can be executed in parallel, the body of research in this category focuses on parallelism and optimization of a single graph processing algorithm, and does not involve multiple algorithms in tandem.

The majority of graph processing prior work exists in this category, and a sizable number of CPU-based graph processing systems have been proposed in recent years, such as Ligra [Shun and Blelloch, 2013], GraphGrind [Sun et al., 2017], Polymer [Park et al., 2022], Gemini [Zhu et al., 2016] Grazelle, GraphMat [Sundaram et al., 2015], GraphLab [Low et al., 2014], Pregel [Malewicz et al., 2010], etc.

These works chiefly focus on data locality improvement [Zhang et al., 2018a, Zhao

Related Work	Architecture	Concurrent	Memory	Data Layout	General Purpose	Scheduling	Distributed Processing	Load Balancing	Programming Model	Automated
Ligra [Shun and Blelloch, 2013]	CPU	X			X					
Polymer [Park et al., 2022]	CPU		X			X				
GraphGrind [Sun et al., 2017]	CPU					X	X			
Gemini [Zhu et al., 2016, Zhu et al., 2016]	CPU						X	X		
GraphMat [Sundaram et al., 2015]	CPU				X				X	
GraphLab [Low et al., 2014],	CPU								X	
Pregel [Malewiczet al., 2010	CPU						X		X	
Seraph [Xue et al., 2014]	CPU	X			X					
GraphM [Zhao et al.,2019]	CPU	X	X							
Multilyra [Mazloumi et al., 2019]	CPU	X					X			
Quegel [Zhang et al., 2016	CPU	X			X					
Congra [Pan and Li, 2017]	CPU	X				X				
Krill [Chen et al., 2021]	CPU	X	X		X				X	
GunRock [Wang et al., 2016]	**GPU**		X	X	X	X				
Boost Graph Library [Siek et al., 2001]	**GPU**						X	X		
Groute [Ben-Nun et al., 2020]	**GPU**								X	
TOTEM [Gharaibeh et al.,2013]	**GPU**		X	X		X				
CuSha [Khorasani et al., 2014]	**GPU**		X	X				X	X	
Multigraph [Hong et al., 2017]	**GPU**			X				X		
Frog [Shi et al.,2017]	**GPU**		X	X						
GraphBLAST [Yang et al., 2019]	**GPU**								X	
Egraph [Zhang et al., 2022]	**GPU**	X							X	
SaGraph [Zhao et al., 2023]	**GPU**	X							X	
Our Framework	**GPU**	X	X		X					X

Table 3.1: Breakdown of existing body of research and their graph processing optimization dimensions on CPUs and GPUs

et al., 2019], graph partitioning for distributed processing [Zhu et al., 2016] [Sun et al., 2017, Malewicz et al., 2010], scheduling optimizations [Wei et al., 2016] and algorithm-level parallelism. As mentioned before, all of these systems are focused on performance optimization of a single graph processing algorithm, running on the CPU. In contrast, our proposed system is both running on the GPU, and focused on multiple graph processing algorithms running in tandem. However, several of the optimizations listed in such work are perpendicular to our work and thus can be incorporated if necessary.

3.2 GPU-based Graph Processing Systems

GPUs can potentially offer a high level of parallelism. However, there are nontrivial challenges in using GPUs for real-world graph processing systems. Unlike CPUs which are perfectly suited for performing low latency tasks , GPUs still lag behind in complex control flow management and flat pageable memory pools. This lack of complex control flow management – inherent to execution of hundreds if not thousands of similar kernels at the same time – makes running large scale graph processing jobs even more challenging on the GPUs, as graph processing commonly requires irregular data access patterns and control flow adjustments.

Despite the aforementioned challenges, there has still been a notable body of existing work dedicated to the use of GPUs for processing large scale graphs. Many of such work are multi-purpose. As such, only the ones exclusively dedicated to graph processing systems are covered in this section.

GunRock [Wang et al., 2016] is perhaps the most well-known graph processing system for GPUs. GunRock focuses on streamlining the development of high-performance graph algorithms through parallelism. Other noteworthy GPU-based graph processing systems that emphasize a single algorithm are works such as the Boost Graph Library [Siek et al., 2001], Groute [Ben-Nun et al., 2020], TOTEM [Gharaibeh et al., 2013], CuSha [Khorasani et al., 2014], Multigraph [Hong et al., 2017], Frog [Shi et al., 2017] and GraphBLAST [Yang et al., 2019]. Each of these systems attempted to address the many challenges that using GPUs in processing

large scale graphs encompasses. Specifically, CuSha, Frog and TOTEM work to address performance problem that irregular data layouts in graph processing systems causes in the GPU. GunRock, CuSha, Frog and the Boost Graph Library attempt to address non-coalesced memory access patterns. Multigraph uses load balancing to improve performance. Groute is a multi-GPU graph processing system, and GraphBlast is high performance linear algebra-based graph processing system on the GPU that attains performance by offering a new programming model.

[Shi et al., 2018] provide a survey of GPU-based graph processing systems. Although such systems utilize parallelism to achieve high performance, all of them focus on optimizing a single graph processing algorithm, in contrast with multiple algorithms running in tandem.

3.3 CPU-based Concurrent Graph Processing Systems

A multitude of frameworks has been developed in recent years to concurrently process multiple graph algorithms on CPUs. Seraph [Xue et al., 2014] is an end-to-end concurrent graph processing framework for processing concurrent graph algorithms, instead of aiming at specific algorithms such as multiple-BFS or PageRank. CGraph [Zhang et al., 2018a] is another system that leverages the locality of multiple concurrent graph algorithms to improve performance. GraphM [Zhao et al., 2019] is also a memory system designed to improve the performance of multiple concurrent graph algorithms on CPUs. Other noteworthy systems in this category include Multilyra [Mazloumi et al., 2019] which offers scalable performance for iterative graph queries, Quegel [Zhang et al., 2016] which offers a new superstep-sharing execution model, and Congra [Pan and Li, 2017] which offers an intelligent graph processing scheduler for better throughput.

Krill [Chen et al., 2021] is a compiler and runtime concurrent graph processing system that leverages both shared data access and processing patterns. Krill also fuses multiple graph algorithms based on a modified frontier-based pull and push engines. It exposes a programming API to developers enabling them to implement customized graph algorithms. Additionally, Krill automatically fuses multiple algo-

rithms together to achieve better overall performance. Even though Krill is designed and implemented on CPUs, it is the closest work to ours. Krill is general purpose, concurrent and automated. It also fuses kernels for memory cohesion. However, as mentioned before, it is CPU-based and works with the assumption of unlimited flat memory models, and as such is not quantitatively comparable to our work.

With the exception of Krill and Seraph, all of the aforementioned concurrent graph processing systems are dependent on the algorithm context, and the fusion method cannot be extended to general graph algorithms.

3.4 GPU-based Concurrent Graph Processing Systems

There has been a limited number of research offering concurrent graph processing solutions on GPUs. Works such as Egraph [Zhang et al., 2022], which offers a Loading-Processing-Switching execution model for timing iterative graph processing jobs and SaGraph [Zhao et al., 2023] which is a concurrent graph processing execution model for temporal graphs are noteworthy examples. However, none of these works offer an automated fusion framework nor they are general purpose. To the best of our knowledge, there are no other automated concurrent graph processing systems targeting GPUs, enabling developers to define their desired graph processing algorithms as kernels on the GPU, and automatically fusing such algorithms to improve performance.

	Job Sets				
Graphs	homo1	homo2	homo3	heter1	heter2
LiveJournal	55%	36%	22%	16%	4.3%
Patents	74%	32%	17%	12.1%	2.8%
Higgs	29%	13.2%	11.7%	8.2%	2.1%
Pokec	1.1%	4%	9%	3.3%	1.7
Youtube	-18%	-11%	-9%	-0.9%	-0.3%
Wiki-Talk	0.99%	0.1%	1.7%	2.3%	2.9%

Table 3.2: Speedup percentages of our framework over Krill [Chen et al., 2021] for different job sets defined in Table 5.3 and graphs presented in Table 5.1, this comparison has not been conducted to evaluate our framework because of the difference in architecture used in Krill(CPU) and our framework(GPU) but to give readers an idea about the difference in execution time between two frameworks. As obvious in the table, our framework has speedup over Krill for most of the job sets and for majority of the graphs, the main reason is that the execution time of the same jobs has been a lot faster on our framework (implemented on GPU) than on Krill (implemented on CPU) mostly before fusion due to the difference between the architectures.

Chapter 4

KERNEL FUSION FRAMEWORK

To enable kernel fusion within GPU, a framework was developed to streamline creation, dispatching and fusion of different GPU jobs. The framework is mostly created in C++, with supporting scripts in other languages.

The first component in the framework is an accurate timer, which is used to profile different sections and activities and provide detailed time measurements. Graphs are represented as adjacency lists using the `AdjacencyGraph` class. They are read from files and stored in CPU and GPU memory as adjacency lists (two arrays for nodes and edges respectively).

GPU jobs inherit from the base class `AbstractJob`, which denotes whether or not a job is active, what is its current frontier, and whether or not it is finished. `AbstractJob` also assists in initialization of jobs, by streamlining memory allocation and movement between CPU and GPU using helper methods.

Each job that inherits from `AbstractJob` needs to explicitly define 4 sections:

1. Auxiliary data (metadata): constants or variables needed by the job, usually involving several integer variables such as starting node index, as well as at least one array the size of the graph (e.g., visited nodes, distance, etc.).

2. Constructor: sets the initial variables of a job, such as the starting node, number of iterations, etc.

3. Initializer: is used to allocate and initialize job data in the GPU and assign their respective data pointers.

4. GPU-specific kernel: used to do the actual computation (in each iteration).

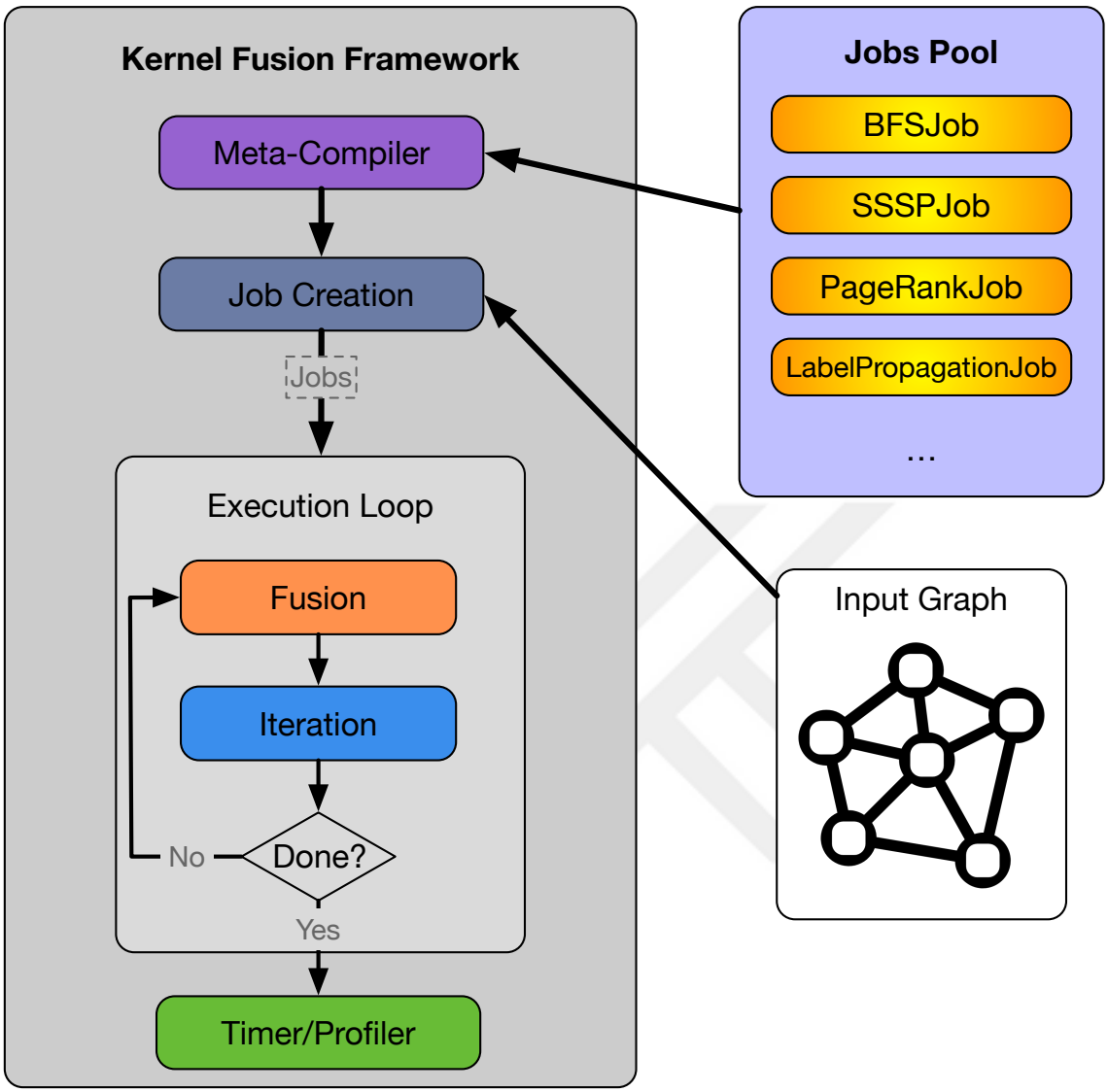


Figure 4.1: The architecture of the kernel fusion framework. The defined jobs in the job pool are compiled by the meta-compiler, and then multiple jobs are created based on an input graph. The created jobs are then passed on to the execution loop, which fuses and runs them until they are all done.

Auxiliary data needs to be defined in a specific order at a certain section of the job definition, as the meta-compiler relies on these definitions to construct the `FusionJob` class (as explained below). The GPU kernel receives the GPU thread id and the input graph. It has to perform both its core computation function, as well as update the job's frontier in the graph (i.e., what the next active nodes are going

to be), and mark whether the job is completed or not.

The framework also includes a `RunJobs` function that iterates over a list of jobs (the list can be dynamically updated at runtime), performs kernel fusion (if enabled) and iterates over all jobs via a GPU kernel called `IterJobs` until all jobs are done. `RunJobs` is also in charge of collecting runtime and performance metrics from CPU, GPU, fusion algorithm, etc.

`IterJobs` is the GPU entry point of the framework, executed once per graph node. It is in charge of polymorphically calling all job kernels for each graph node, and passing them their respective metadata. `IterJobs` also checks whether jobs are still active and whether they are still not finished, before dispatching them. All jobs that are active and not completed will be dispatched, in order, during `IterJobs`.

Finally, the framework includes `Fusion`, which is in charge of fusing job kernels based on their data access patterns to improve performance.

```
❶ AdjacencyGraph graph = LoadGraph(file);
❷ FusionJobs jobs[] = CreateJobs(job_count, graph);
❸ while (not AllJobsDone()):
    ❹ if (isFusionEnabled) Fusion(jobs);
    ❺ IterJobs<<<graph.size/1024, 1024>>>(jobs, graph);

__global__ IterJobs(FusionJob jobs[], AdjacencyGraph graph):
  ❻ id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id >= graph.size) return;

  for (i = 0; i < jobs.count; i++):
    ❼ if (not jobs[i].active) continue;
    ❽ if (jobs[i].done) continue;
    ❾ if (not jobs[i].frontier[id]) continue;
    ❿ // MetaCompiler Generated Below:
    if (jobs[i].job_type == JobTypes::BFS)
      ((BFSJob)jobs[i]).iter(id, graph);
    else if (jobs[i].job_type == JobTypes::SSSP)
      ((SSSPJob)jobs[i]).iter(id, graph);
    else if (jobs[i].job_type == JobTypes::PageRank)
      ((PageRankJob)jobs[i]).iter(id, graph);
    else
      print("Unsupported job type!");
```

Listing 4.1: The simplified overall flow of the fusion framework.

Figure 4.1 provides an overview of the described architecture, while Listing 4.1 provides an overview of the framework and how it works. Line 4.1.❶ defines the input graph, loads it from the input file (into both the CPU and the GPU), and assigns it to a variable. Line 4.1.❷ creates several jobs based on the defined criteria. Each created job is automatically initialized by allocating necessary memories, and setting initial metadata such as starting node, distance arrays, etc. Line 4.1.❸ is the start of simplified `RunJobs` function, where jobs are executed until all of them are done. Line 4.1.❹ performs kernel fusion for the jobs (if enabled). The result of this fusion is that some jobs will be active (for the current iteration), while others will be inactive. Line 4.1.❺ finally launched the GPU kernel `IterJobs` that runs

the jobs in the GPU.

The rest of the listing covers the IterJobs GPU kernel. This kernel is executed as many times as there are nodes in the input graph. Line 4.1.❻ assigns the id of the current GPU thread to the variable `id`. The subsequent line ensures that `id` is not larger than the number of nodes in the graph. Line 4.1.❼ – the starting line of the loop that is executed as many times as the number of jobs – makes sure that the job is active in this iteration. Without fusion, all jobs are always active. With fusion, only jobs that are selected by the fusion algorithm are active. Line 4.1.❽ skips any job that is already finished. Line 4.1.❾ makes sure that the current job in the current execution for the current thread id has an active frontier, i.e., nodes that are actively being worked on. Line 4.1.❿ and subsequent lines are auto-generated by the meta-compiler to enable polymorphism, and run the job-specific iteration function. This is further expanded on in Section 4.1.

4.1 Polymorphism

Because the framework needs to be able to run different jobs at runtime (as a list of jobs), and because it is a compiled C++ program, it needs to utilize polymorphism. However, this polymorphism is partially on the CPU (initialization, fusion) and partially on the GPU (kernels, fusion).

This work investigated three avenues for implementation of this polymorphism: C++ virtual functions and inheritance (native polymorphism), Runtime Type-Information (RTTI) based polymorphism, i.e., adding extra variables and code to jobs that defines their runtime types and behaviors, and meta-compiler static polymorphism, i.e., generating code based on available types and only passing minimal job type information at runtime via switch statements.

Native polymorphism enjoys somewhat limited support in CUDA. Specifically, it requires objects to be allocated on CUDA heap directly, otherwise virtual function tables will not be *on device* (i.e., in GPU memory). However, the default heap is quite limited, and can only be resized once per program, to a maximum of about 80% of GPU memory, i.e., it cannot be sized down later. Furthermore, our evaluations

show that there is a significant performance hit for using native C++ polymorphism in CUDA. For these reasons, we decided to use other alternatives.

RTTI-based polymorphism requires specific compiler flags and passing of additional variables and data to the GPU, as well as allocation of objects on GPU heap within GPU code. Due to these limitations, this approach was also not desirable.

Finally, it was decided to use a meta-compiler that adds small sections of code based on available GPU jobs and then compiles them to native code.

4.1.1 Meta-Compiler

The meta-compiler scans the code for all defined jobs that inherit from `AbstractJob`, and creates a list of available jobs as a C++ Enum. For example:

```
❶ enum JobTypes {
    Abstract,
    BFS,
    SSSP,
    PageRank,
  };

❷ class BFSJob: public AbstractJob {
 public:
      JobTypes job_type; // Holds type of job for static polymorphism.
      BFSJob(u64 root) : root(root) {
          job_type = BFS;
      }
```

Listing 4.2: The C++ enum of all jobs generated by the meta-compiler, as well as its usage in constructor of jobs to define their runtime type.

This enum is then used in each class's constructor to explicitly set its type in a member variable called `job_type`, as apparent in 4.2.❷.

The meta-compiler subsequently adds a small section of switch statements inside `IterJobs` that dispatches the appropriate job kernel based on the type of the job instance (Listing 4.1 line ⑩).

Finally, and most importantly, the meta-compiler needs to create a new `FusionJob`

```
class PageRankJob: public AbstractJob {
       u64 root;
       u64 *visited;
       double* pagerank;
       u64 max_iterations;
...

class SSSPJob: public AbstractJob {
       u64 root;
       i64* distance;
...

class BFSJob: public AbstractJob {
  u64 root;
  i64* parents;
...
```

Listing 4.3: Different computational jobs defined in the framework, and their respective metadata.

structure, that includes metadata of all available jobs in as little memory as possible, as this data is allocated and passed to the GPU in bulk for all jobs at the same time (for performance reasons).

4.1.2 FusionJob Structure

To generate the code for FusionJob structure, the meta-compiler needs to scan all available jobs and determine which metadata each one uses. The order of usage of this metadata, as well as their variable names and variable types are then used to create a class called `FusionJob` that uses C++ unions to minimize memory footprint, while allowing access to all these variables and their respective types in each job.

For example, consider the three jobs PageRank, SSSP and BFS with the following metadata: Which turns into the following `FusionJob` structure by the meta-compiler:

```
struct FusionJob: public AbstractJob {
    u64 root;
    union {
        i64* parents;
        i64* distance;
        u64* visited;
    };
    double* pagerank;
    u64 max_iterations;
```

Listing 4.4: The resulting FusionJob sructure generated by the meta-compiler that includes all member variables used by different jobs, in unions.

The structure in this example uses only 4x8 = 32 bytes of memory per job, and allows each job to effectively access variables it assumes exist in it. Additionally, because it observes the ordering of the defined variables in the jobs themselves, low-level memory copy operations of lists of jobs will preserve validity of jobs when they are passed along between the GPU and the CPU and within the CPU and the GPU.

4.2 Kernel fusion

As depicted in Figure 4.1, before each iteration, the framework performs kernel fusion (if enabled). Kernel fusion can be disabled to compare runtime performance with and without fusion. Additionally, the framework distinctively reports total job completion time, time spent on GPU, and time spent on fusion, as well as a complete profiling of time spent for each of the fusion steps.

4.2.1 Algorithm

The fusion algorithm finds the shared frontier of all jobs, i.e., the union of all the nodes that all jobs will work on in the next iteration. It then finds the subset of this shared frontier that is maximally used by all remaining jobs. Finally, the algorithm marks jobs that will work on this maximal shared frontier in the next iteration, as active, and all other jobs as inactive. This algorithm ensures that memory accesses

```
bool fusion(const int job_count, const int graph_size, FusionJob* d_jobs) {
  int shared_frontier[graph_size]; // Number of jobs active on each node.

  device_allocate_and_copy(shared_frontier);
  fusion_kernel_phase1<<<graph_size, graph_size/1024>>>(job_count, graph_size,
    d_jobs, d_shared_frontier, d_job_frontier_size);
  device_copy_back(shared_frontier);

  // Find the frontier vertex with maximum jobs:
  int max_index = 0;
  for (int i=0; i<graph_size; ++i)
    if (shared_frontier[i] > shared_frontier[max_index])
      max_index = i;

  fusion_kernel_phase2<<<1, 1>>>(job_count, graph_size, max_index, d_jobs);
}
```

Listing 4.5: Pseudo-code for the fusion algorithm.

to nodes on this maximal shared frontier will be shared by several jobs, resulting in effective caching in memory access which can significantly speedup job execution.

Listing 4.5 shows the pseudo-code of the fusion algorithm as implemented in the `fusion` function. The function receives number of jobs (`job_count`), size of the graph (`graph_size`) and the list of jobs on the GPU (`d_jobs`).

In the first phase of fusion, it calculates the shared frontier of all jobs in the GPU, by basically atomically incrementing the number of jobs working on each node in the graph on the shared `shared_frontier` array. After this phase, each element of `shared_frontier` array will include the number of jobs having that node in their current, active frontier.

Afterwards, the element in this array with the highest number is found. This is the node that has the highest number of jobs actively working on it, and the fusion algorithm needs to keep all the jobs working on this node active, while disabling all other jobs. Enabling and disabling of jobs is done in phase 2 of the function,

directly in the GPU, as the job structures (`d_jobs`) reside inside the GPU.

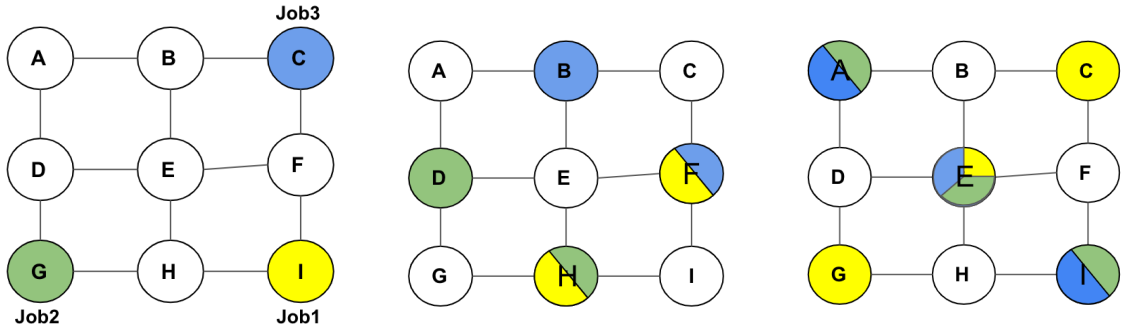


Figure 4.2: The expanding frontier of 3 jobs running in parallel.

Figure 4.2a shows 3 jobs traversing the same graph from three different starting nodes, node I, G and C, respectively. In the next iteration (Figure 4.2b), the jobs have expanded to access nodes (F, H) for job 1, (D, H) for job 2 and (B, F) for job 3. Notice that no node is shared between all three jobs, whereas node H is shared between job 1 and job 2, and node F is shared between jobs 1 and 3. As such, fusion can deactivate job 3, and run jobs 1 and 2, making them access node H simultaneously, resulting in one less uncached memory access.

In the next iteration (Figure 4.2c), each job has an expanding frontier that encompasses 3 nodes. This is a typical trend in graph traversal, in the first few iterations, the frontiers are quite small, but they exponentially expand to include a significant portion of the entire graph, and then gradually dwindle in size to reach zero once the traversal is completed. As apparent from the Figure, in the third iteration, nodes (A, E, I) are shared between jobs 2 and 3, while node E is shared between all three jobs. If the fusion algorithm picks jobs 2 and 3 to run, there will be 3 memory accesses for traversing 3 nodes by 2 jobs (i.e., 6 node accesses), pushing execution of job 1 to the next iteration with its own distinct 3 memory accesses; whereas if the fusion algorithm picks all 3 jobs, there will be 5 memory accesses for traversing 5 nodes by 3 jobs (i.e., 9 node accesses). The latter selection results in a total of 5 memory accesses for completion of iteration 3 of all jobs, in contrast with

6 memory accesses in the former.

Our evaluations show that accessing the same memory regions by concurrent jobs can result up to 500 times faster runtime compared to sporadic memory access by different jobs. This speedup is achieved by running all jobs in the same order, starting from the same node and traversing the graph in identical patterns, in contrast with each job starting at a different point or performing a different traversal.

Finding the maximal shared frontier is a computationally intensive algorithm. Frontiers of all jobs need to be scanned first – each of which is potentially as large as the entire graph – to find the shared frontier among all jobs. To find the *maximal* shared frontier within the previously discovered shared frontier, the fusion algorithm needs to count how many times each node in the shared frontier is accessed by each job. Once the nodes with maximum access count are discovered, jobs that have these nodes in their frontier need to be listed, and subsequently marked as active (while all other jobs are deactivated).

As such, the framework implements the part of the fusion algorithm that finds the maximal shared frontier, inside the GPU itself (to follow a more CPU-independent execution model [Ismayilov et al., 2023]), avoiding the need to copy frontiers of each job back to the CPU, and reducing the footprint of fusion from around 10% of job execution time to between 1% to 2% of job execution time.

Chapter 5

EVALUATION

In this section, datasets, environment setup and job lists are described, then multiple experiments are performed to evaluate the effectiveness of the proposed automated kernel fusion framework.

5.1 Experiment Setup

5.1.1 Graph Dataset

The graph dataset used for evaluations is listed in Table 5.1. Six widely used graphs – with varying sizes and structures – were selected. All of these graphs are extensively used in previous works [Chen et al., 2021, Pan and Li, 2017] , and range from 500k nodes to 5 million nodes in size.

The *Live Journal Social Network Graph* [Leskovec and Krevl, 2014] (`LiveJournal`), is a directed graph consisting of about 5 million nodes and about 70 million edges. All of the nodes are in one connected component, and the largest strongly connected component (SCC) has 4 million nodes. The graph has a diameter of 16, i.e., the longest shortest-path throughout the graph is 16 edges. The LiveJournal graph has a very large number of edges, and algorithms that are edge-heavy are expected to perform slower on it. Additionally, it is the second-largest graph in our dataset.

The *US Patent Citation Network* [Leskovec and Krevl, 2014] (`Patents`), is an undirected graph of 6 million nodes and 16 million edges, with a diameter of 22 and consisting of one SCC. `Patents` is the largest graph in our dataset based on the number of nodes, but has much fewer edges than `LiveJournal`.

The *Higgs Twitter Dataset* [Leskovec and Krevl, 2014] (`Higgs`), has been built by monitoring the spreading processes on Twitter related to the announcement of the discovery of Higgs boson [Weinberg, 1976]. The graph has 500k nodes, 15m

edges, a diameter of 9, and 360k nodes in the largest SCC. This is the smallest graph in our dataset, albeit with a proportionally large number of edges (the same number of edges as `Patents`, less than 10x the number of nodes).

The *Pokec Social Network Graph* [Leskovec and Krevl, 2014] (`Pokec`), is based on the most popular social network in Slovakia. The graph has 1.6m nodes, 3m edges and a diameter of 11, and 1.3m nodes in its largest SCC. The graph has about 3 million edges in contrast with 1.5 million nodes, meaning that on average each nodes has only 2 edges.

The *Youtube Social Network* [Leskovec and Krevl, 2014] (`Youtube`) graph is based on the social network of people forming communities on YouTube. The graph consists of 1m nodes and 3m edges, and has a diameter of 20, with one SCC. Similarly to `pokec`, this graph has 3 edges per node on average.

Finally, the *Wikipedia Talk Network* [Leskovec and Krevl, 2014] (`Wiki-Talk`) graph includes the communications among Wikipedia editors as recorded in Wikipedia Talk pages. The graph contains about 2.5m nodes and 5 million edges, and has a diameter of 9, with 1 million nodes in its largest SCC.

Note that larger graphs may result in reduced maximum parallel feasible jobs, as many jobs require metadata that grows proportionally to the size of the graph. For example, if the PageRank job needs to maintain 16 bytes of metadata (one integer and one double precision floating point) per node in the graph, each PageRank job running on a graph that has 5 million nodes would take 80MB of GPU memory. Running 100 similar jobs in parallel would need 8GB of GPU memory just for retaining the job metadata. It is for this reason that several experiments – especially those that use more complex jobs – have a lower number of maximum parallel jobs (job count) in contrast with experiments that solely incorporate simple jobs.

5.1.2 Graph Processing Algorithms

Four popular graph processing algorithms – widely used by previous work and in the industry – were selected and implemented on top of the framework as jobs. The implementations are straightforward and not particularly optimized, following the

	Graphs	Vertex Count	Edge Count	Diameter
1	LiveJournal	4,847,571	68,993,773	16
2	Patents	6,009,555	16,518,948	22
3	Higgs	456,631	14,855,875	9
4	Pokec	1,632,803	30,622,564	11
5	Youtube	1,134,890	2,987,624	20
6	Wiki-Talk	2,394,385	5,021,410	9

Table 5.1: The list of graphs used in evaluations dataset.

	Algorithm	**Description**
1	BFS	Breadth first search
2	SSSP	Single-source shortest path
3	PageRank (PR)	Measuring relative importance
4	Label Propagation (LP)	Labelling and clustering of a graph

Table 5.2: Graph processing algorithms selected and implemented as jobs for evaluation.

job definition patterns defined in the framework. These algorithms include Breadth-First Search (BFS), Bellman-Ford Single-Source Shortest Path (SSSP), PageRank (PR) and Label Propagation (LP) algorithm, as listed in Table 5.2.

5.1.3 Experiment Job Sets

To demonstrate the efficacy of the automated kernel fusion framework in different work load scenarios, 10 different job sets were devised, each of which containing a different combination of algorithms. These job sets can be observed in Table 5.3. Job sets can be divided into two categories, *homogeneous*, where all parallel jobs are of the same algorithm (although with different parameters such as starting nodes), and *heterogeneous*, where different parallel jobs are of different algorithms. For the

Job Set	**Algorithms**
Homogeneous 1	BFS x N
Homogeneous 2	SSSP x N
Homogeneous 3	PR x N
Homogeneous 4	LP x N
Heterogeneous 1	{BFS, SSSP} x N
Heterogeneous 2	{BFS, SSSP, PR} x N
Heterogeneous 3	{LP, BFS} x N
Heterogeneous 4	{LP, SSSP} x N
Heterogeneous 5	{BFS, SSSP, LP} x N
Heterogeneous 6	{BFS, SSSP, PR, LP} x N

Table 5.3: Job sets used in experiments and evaluations. Homogeneous job sets include varying number of the same graph algorithm, for example multiple BFS jobs and heterogeneous job sets include a mixture of different graph algorithms, for example heterogeneous job set 1 include multiple BFS and multiple SSSP jobs.

sake of simplicity, all heterogeneous job sets combine the selected algorithms in a round-robin fashion, e.g., a heterogeneous job set of SSSP, PageRank and BFS, would have the first job of SSSP algorithm, the second job of PageRank algorithm, the third job of BFS algorithm, the fourth job of SSSP algorithm, the fifth job of PageRank algorithm, and so forth.

As this work is the first to automatically fuse graph processing algorithms together on a GPU, to show the performance gain of the proposed fusion method, all evaluations compare the same execution with and without the fusion algorithm. To this end, each experiment is repeated once with fusion enabled, and once with fusion disabled.

As previously mentioned, all experiments are also performed on each platform with an increasing job count that goes from 1 parallel job up to 200. However, not all platforms have enough GPU memory to contain the metadata necessary for 200

	Name	**GPU + RAM**	**CPU + RAM**
1	AMD Rome Workstation	NVIDIA RTX A5000 48GB	AMD Zen 2 512GB
2	AMD with A100 Workstation	NVIDIA A100 40GB	AMD Zen 2 512GB

Table 5.4: Platforms used for evaluations.

parallel jobs. In those cases, the job count is incremented until the GPU memory is exhausted. In the evaluations, the platform with the smallest GPU memory (8GB) is able to run 65 instances of the most demanding job (memory-wise) in parallel. Many experiments were also repeated to ensure spikes and anomalies are not temporal.

5.1.4 Evaluation Platforms

Two distinct platforms were used to perform our extensive evaluations.

The first, the *AMD Rome Workstation* a scientific computation server containing one instance of NVIDIA RTX A5000 with 48 GB of GPU RAM and AMD Zen 2 microarchitecture with 512 GB of RAM.

The second, the *AMD with A100 Workstation* a scientific computation server containing one instance of NVIDIA A100 GPU with 40 GB of GPU RAM, AMD Zen 2 microarchitecture with 512 GB of RAM.

NVIDIA driver version 510.47.03 [Nvidia Corporation, 2022c] and CUDA version 12.1 [Nvidia Corporation, 2022a] were used. The same open-sourced source code was used on all instances, and compiled without any flags, which defaults to the `-O3` optimization flag by default on the NVIDIA Compiler (`nvcc`) [Nvidia Corporation, 2022b].

Table 5.4 summarizes the platforms used in experiments and evaluations.

5.2 Performance Evaluation

In this section, several experiments used to analyze the performance of the proposed automated fusion method in different job sets are presented.

The result of running time of each job set on different graphs on platform 1

are depicted in Figures 5.1, 5.2, 5.3 and 5.4. Each graph in Figures 5.1 and 5.3 depicts runtime of one job set on all input graphs with increasing job counts, with and without fusion. Dotted lines plot runtime without fusion, while dashed lines demonstrate runtime with fusion.

5.2.1 Homogeneous Job Sets Performance

As immediately observable in the figures, not all runtime with fusion are better than their counterparts without fusion. This slowdown is specially pronounced when job count is small (Figure 5.1a). Additionally, slowdown is observed in certain algorithms such as Label Propagation on graphs that have a significantly large number of edges compared to the number of nodes, such as Higgs, Youtube and LiveJournal (Figure 5.1d).

As apparent in the figures, the runtime curve for fused jobs are generally below the curve for jobs without fusion for almost all graphs and algorithms in homogeneous job sets. This result is expected, and can be because similar jobs are more likely to have similar access patterns [Chen et al., 2021], leading to similar frontier sets (both in size and shape) resulting in reduced memory access and improved runtime.

A notable exception to this improved performance is the Label Propagation algorithm in Figure 5.1d. As apparent from the figure, most fusion curves have *decreased* performance in contrast with non-fusion curves, with the notable exception of the Youtube graph. This slowdown is because the Label Propagation algorithm needs to iterate all edges of each frontier node, to find the intersection of that node's neighbors with the starting node. As each node may have hundreds of thousands of edges, cache utilization quickly becomes ineffective when many concurrent LP jobs are executed, resulting in degraded performance.

Another visible anomaly is the *spikes* in runtime performance of Label Propagation on the Youtube graph (Figure 5.1d). The fusion curve seems to be generally below the non-fusion curve, but has several spikes that sometimes go above the non-fusion curve (circa job count 70). Upon further inspection, similar trends can

be observed in Figure 5.1a and Figure 5.1b, with respect to both Youtube and Wiki-Talk graphs. Our hypothesis is that the shared frontier becomes too large to effectively utilize GPU memory cache at this job count, and as such fusion adds overhead instead of improving performance. To ascertain that these anomalous spikes are not due to temporal evaluation results, these experiments were repeated three more times.

Figure 5.2 illustrates fusion speedup in a bar chart where the performance results are depicted before and after fusion for varying job counts, for all input graphs.

As apparent from the figure, for job counts below 25, there is usually negative speedup, as there isn't a significant shared frontier to fuse jobs together. However, as job count goes beyond 25 and towards 50, speedup increases significantly. Around 55 job count is where speedup starts to dip, although the average speedup is still about 10% thereafter.

The anomalies discussed earlier also show themselves in speedup graphs (Figures 5.2c and 5.2d). Aside from the Youtube graph, Label Propagation jobs generally result in negative speedups, while PageRank is the opposite, with positive speedups for job counts above 25 for all graphs, and negative speedups for job counts above 65 only for the Youtube graph.

5.2.2 Heterogeneous Job Sets Performance

Figure 5.3 illustrates runtime performance of heterogeneous job sets on all input graphs with increasing job counts. Similarly to Figure 5.1, most fusion runtime curves are below their non-fusion counterparts, demonstrating that fusion is reducing runtime and increasing performance in general.

Notable exceptions are again the Youtube fusion graph, which in most charts fluctuates above and below its non-fusion counterpart. This fluctuation is most likely due to cache-aligning issues caused by the large number of edges in the Youtube graph, and can potentially be remedied in future work.

Figure 5.3f depicts experiments including all algorithms in equal number of jobs. As apparent from the figure, about half of the input graphs result in noticeable

performance gain, while another half of the input graphs result in slowdowns. The speedup and slowdowns are also not definitive in each graph, as increasing job count generally results in better speedup. This trend is further visible in Figure 5.4f, where speedups are more pronounced until 50 parallel jobs, and then slowdowns gradually mount up. Certain graphs such as Wiki-Talk continue to have speedups all the way through 100 parallel jobs, whereas graphs such as Higgs have consistent slowdowns.

Figure 5.4 illustrates fusion speedup in a bar chart where the performance results are depicted before and after fusion for varying job counts, for all input graphs. As observable in Figure 5.4a, 5.4b and 5.4c, there is a general speedup with the sporadic exception of the Youtube graph. On the flip side, Figure 5.4d shows general speedups for all graphs, while Figure 5.4f shows consistent speedups, even for the Youtube graph. On average, hetero job sets experience a speedup of 4.9% with fusion.

5.2.3 Fusion Overhead

To measure overhead incurred by fusion, it is first important to analyze how fusion can cause overhead. There are two aspects by which fusion adds overhead to the running time of jobs. The first is the running time of the fusion algorithm itself. The proposed framework uses two GPU kernels and a minimal CPU function – all three of which are heavily optimized – to perform kernel fusion. As such, the overhead caused by the fusion algorithm is generally negligible, averaging 90 milliseconds in contrast with an average running time of 9.57 seconds on the first platform and 99 milliseconds in contrast with an average running time of 6.81 seconds on the second platform, resulting in 3.3%, 0.9% and 1.5% average fusion algorithm overheads.

Note that as apparent from Figure 5.5 and Figure 5.6, as the number of jobs and the overall execution time increases, the fusion algorithm overhead as a proportion reduces. This proportional reduction is the reason behind an average of 3.3% average fusion algorithm overhead on platform 1, that reduces to less than 1% in platform 2, which has a larger memory and supports many more parallel jobs.

The second aspect by which fusion adds overhead to the running time is by increasing the number of iterations required to finish all jobs, as certain jobs are de-

activated and stalled by the fusion algorithm to maximize the active shared frontier. For example, running 100 homogeneous SSSP jobs on the `Youtube` graph with fusion, requires 42 iterations to complete, whereas the same job without fusion finishes in only 21 iterations. This overhead is significantly harder to measure and predict, as it is based on the shared frontier of jobs at runtime, depending on the runtime parameters of each job.

Figures 5.5 and 5.6 depicts these two overheads caused by fusion, in a selection of job sets, graphs and platforms. Each blue bar in each figure depicts the running time of one job with fusion disabled, side-by-side with the orange/red bar depicting the same job's runtime with fusion enabled. The fusion-enabled bar has separate parts for base execution time (including extra iterations caused by the fusion) as well as the fusion algorithm runtime itself, colored in the diagrams with orange and red respectively.

There are several observations to be made from the figures. First, the fusion algorithm overhead is very small. In Figure 5.5a this overhead can get up to 10% due to the fast baseline execution of the BFS algorithm and the large number of edges in the Youtube graph which results in a rapidly expanding shared frontier. Alternatively, in Figure 5.5f this overhead is down to 2%.

Second, as apparent in Figure 5.5c, the baseline execution time of both fused and non-fused jobs are pretty similar, resulting in the fusion overhead causing an overall slowdown, whereas in Figures 5.5b, 5.5d and 5.5e the base execution time is reduced so much due to fusion that the fusion algorithm overhead does not negate the achieved speedup.

Third, in Figure 5.5a for job counts above 60, despite base execution time of fused jobs being noticeably faster than that of the non-fused jobs, the fusion overhead is pronounced enough to result in an overall slowdown in certain experiments.

Fourth, as observable in Figure 5.6a and 5.6b, in heterogeneous workloads, the fusion algorithm overhead is even less significant, as the baseline execution time of the jobs becomes more pronounced.

Finally, as apparent in Figures 5.6d and 5.6e, as job counts increase, the baseline execution time of fused jobs reduces significantly, while the fusion algorithm

overhead grows only slightly, resulting in an overall increasing trend in performance gain.

5.3 Case Study

In this section, we turn our focus to a case study that includes only platform 1, and the BFS, SSSP and PageRank algorithms, evaluated homogeneously and then heterogeneously, resulting in a total of 4 experiment sets. These experiments are all performed on the Wiki-Talk graph, as it has a reasonable size as well as a reasonable number of edges with respect to nodes. The point of this case study is to focus more on the trends observed in fusion. As such, the fusion framework is compiled in debug mode for these experiments (to provide more fine-grained performance metrics).

All jobs were run with and without fusion, with a job count of 1 to 100. Certain jobs such as PageRank and Hetero needed additional metadata structures, and as such 100 instances of them could not fit on the GPU memory at once. Specifically, PageRank job count was tested from 1 to 65, and anything beyond would not fit in GPU memory. Hetero was run with the job count of 1 to 93, for similar reasons. Each experiment was repeated several times to ensure spikes and anomalies are not temporal.

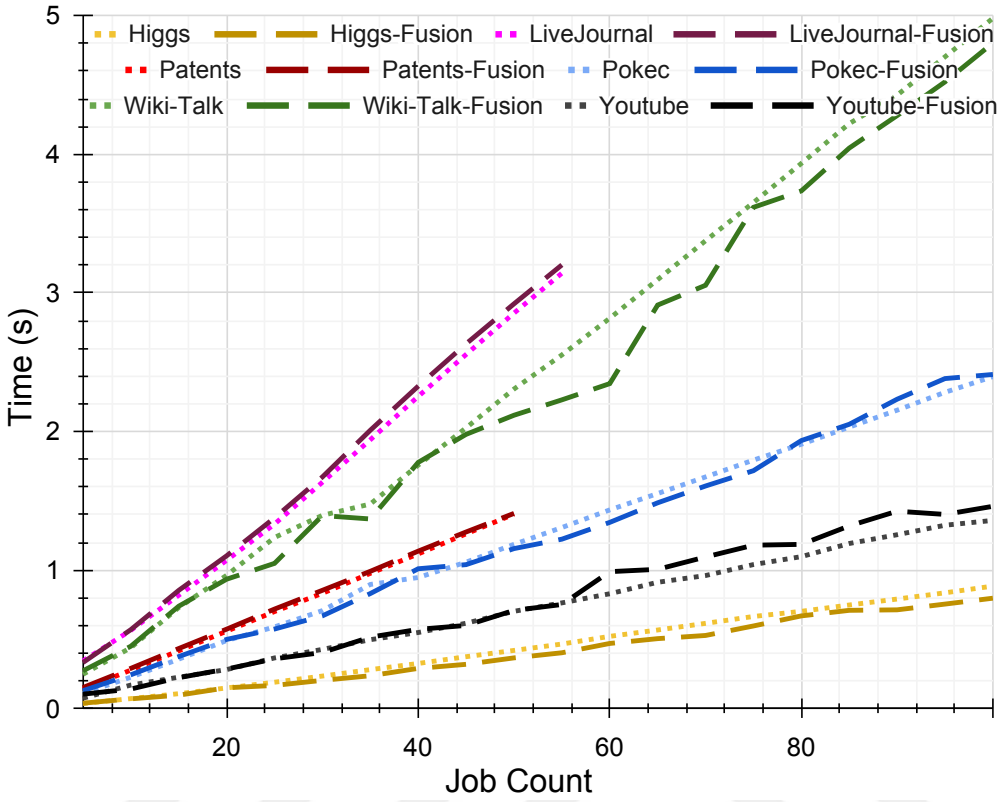
Figure 5.7a shows comparison of each job along with its fusion-enabled counterpart in running time, for job count varying between 1 and 100. As apparent from the figure, certain job counts result in runtime spikes with fusion, which is most likely due to suboptimal shared frontier patterns (as discussed earlier). Figure 5.7b shows zoomed-in version of Figure 5.7a, for job count ranging from 1 to 25. As apparent from the figure, a clear divergence between runtime of fused and non-fused instances of the jobs appears for job counts greater than 15, with the exception of PageRank. This divergence demonstrates that fusion is most significant as job count increases, and may incur additional overhead for small job counts.

These evaluations show a general speedup of around 10% when the job counts are reasonably high. As apparent in Figure 5.8, for low job counts (under 15 parallel jobs) there are sometimes negative speedup (i.e., overhead) due to fusion. On

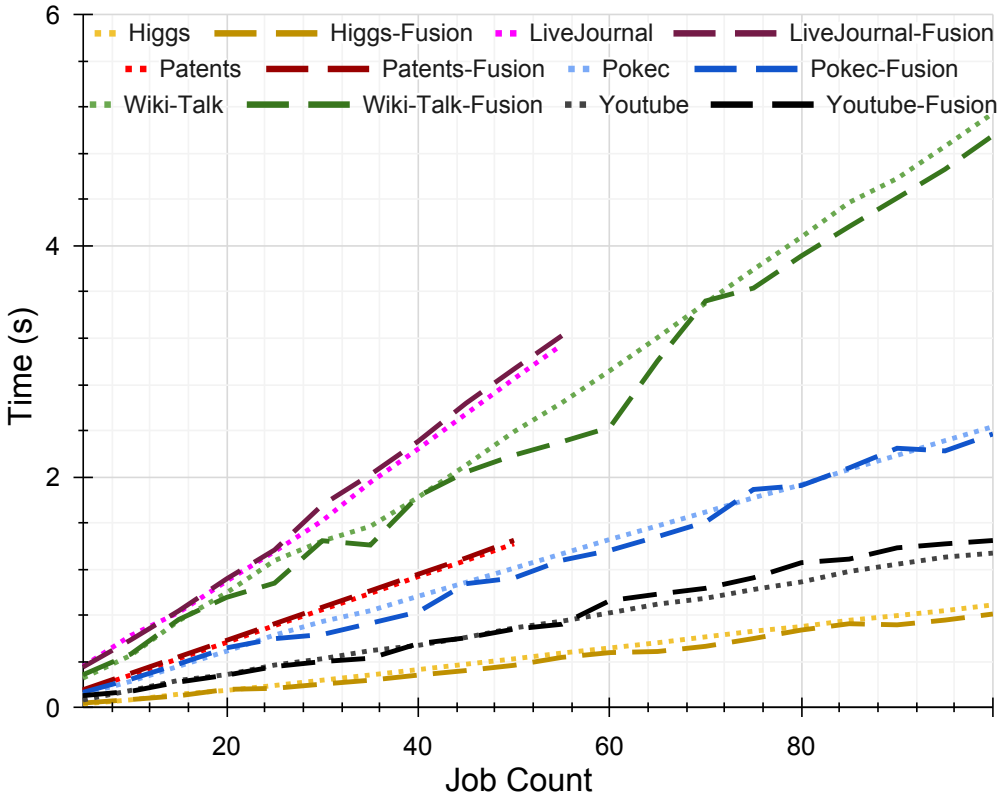
average, fusion results in speedups of 2.43% on PageRank, 4.80% on Hetero, 10.19% on SSSP and 9.52% on BFS (including negative speedups) and speedups of 7.19% on PageRank, 7.86% on heterogeneous execution, 13.05% on SSSP and 11.77% on BFS (for job counts above 50).

The fusion algorithm's overhead itself is between 1% to 2% on all jobs and all job counts above 10. For job counts below 10, fusion overhead can be as high as 30% for a single job, 6% for two parallel jobs, 5% for three parallel jobs, etc.

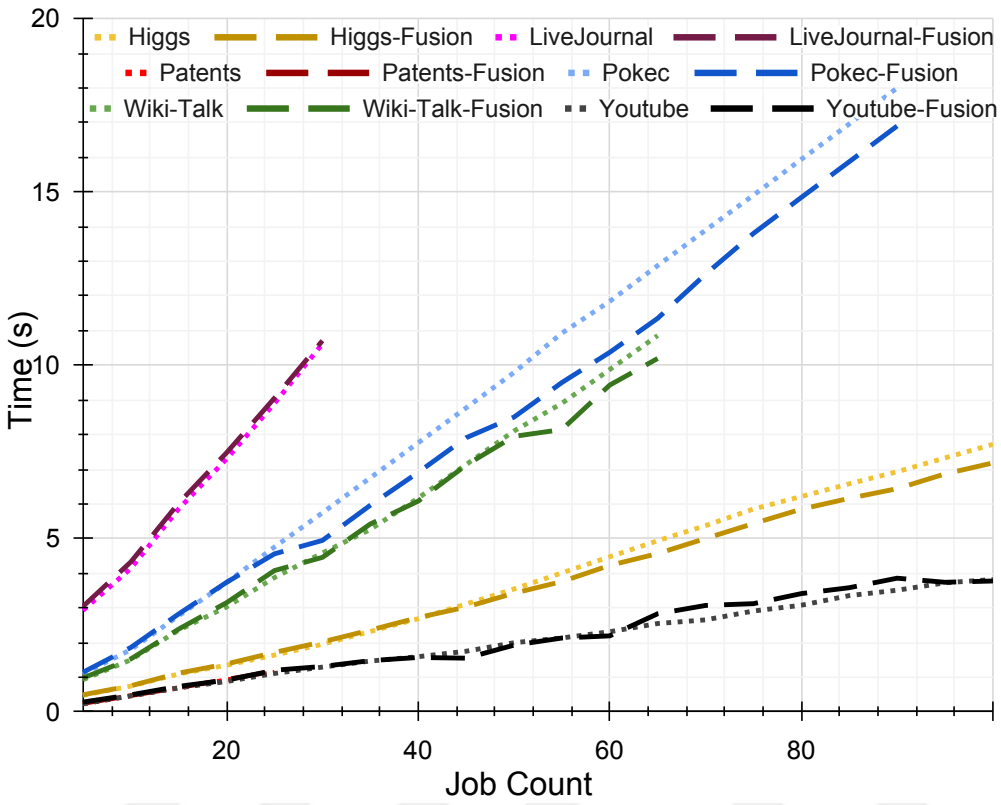
The most significant factor contributing to fusion speedup is memory access patterns in the job. Jobs that have more homogeneous memory access patterns (such as SSSP) can benefit greatly from shared frontiers, while jobs that have many indirect memory accesses (such as PageRank) get less speedup due to fusion.


Higgs
Higgs-Fusion
LiveJournal
LiveJournal-Fusion
Patents
Patents-Fusion
Pokec
Pokec-Fusion
Wiki-Talk
Wiki-Talk-Fusion
Youtube
Youtube-Fusion
Time (s)
Job Count
0
1
2
3
4
5
20
40
60
80

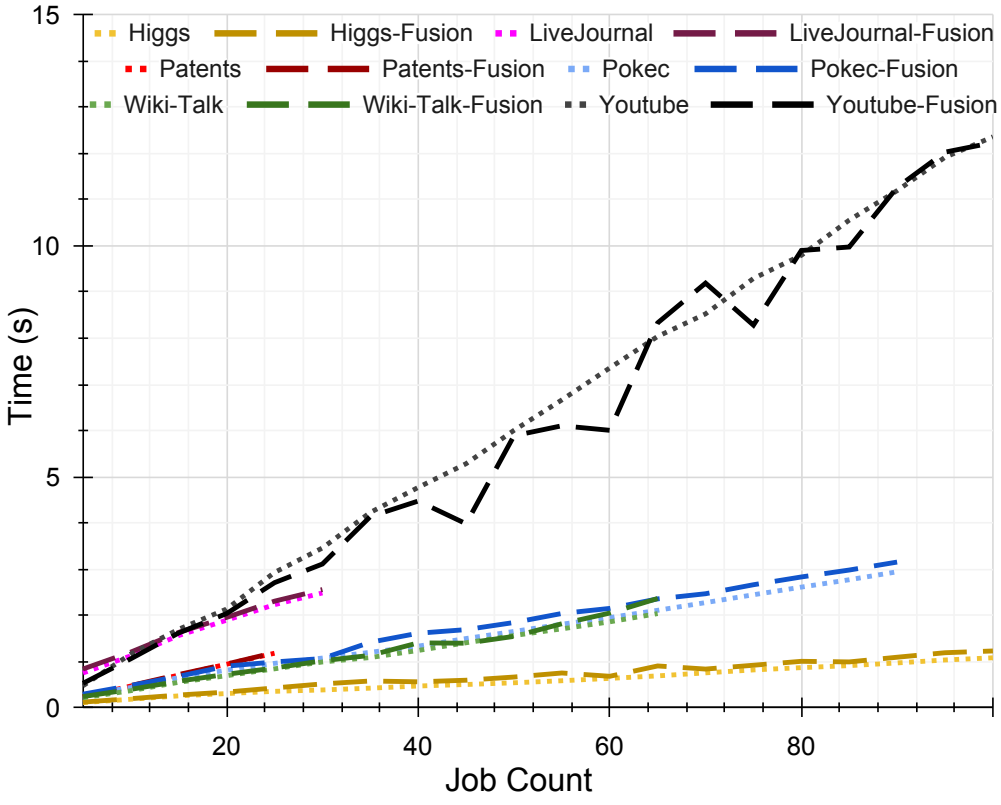
(a) BFS


Higgs
Higgs-Fusion
LiveJournal
LiveJournal-Fusion
Patents
Patents-Fusion
Pokec
Pokec-Fusion
Wiki-Talk
Wiki-Talk-Fusion
Youtube
Youtube-Fusion
Time (s)
Job Count
0
2
4
6
20
40
60
80

(b) SSSP

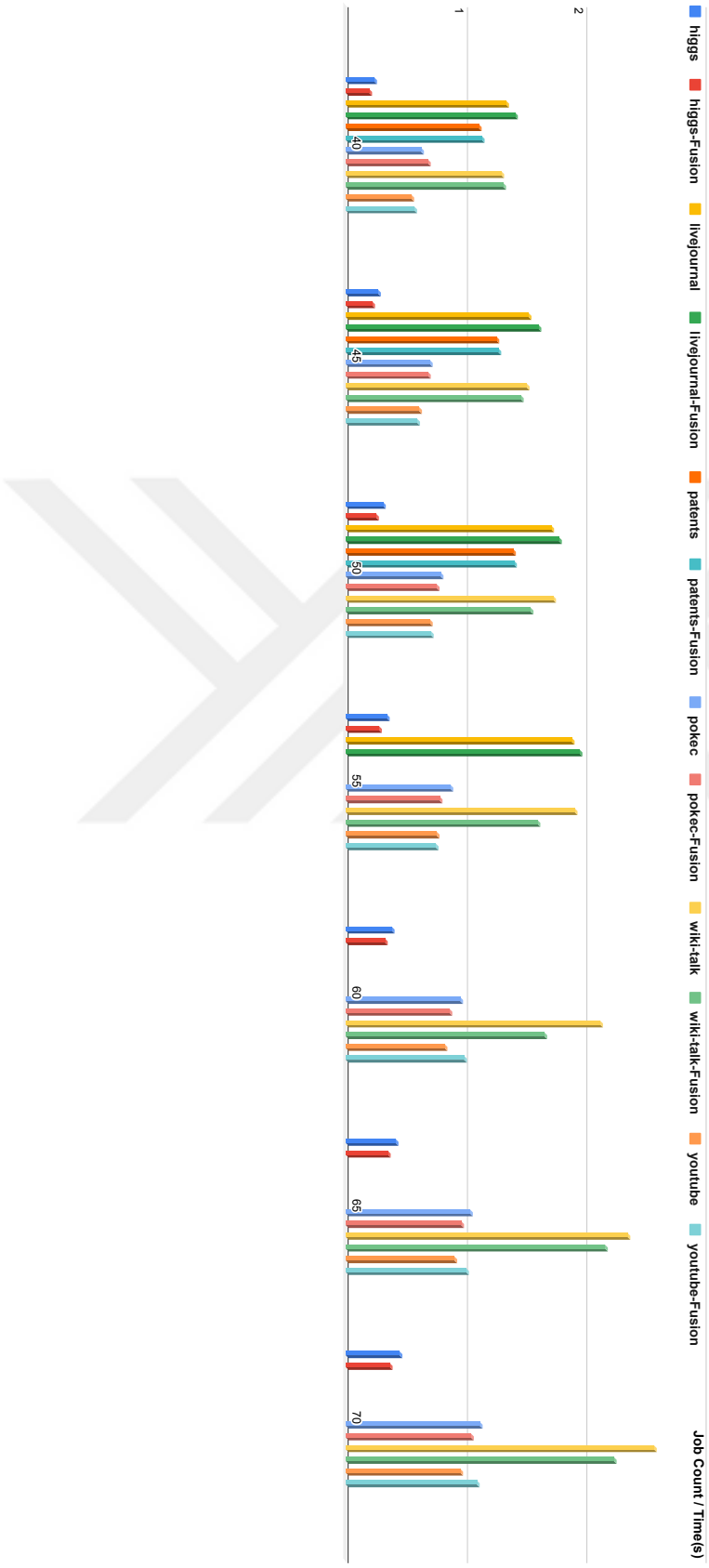


(c) PageRank

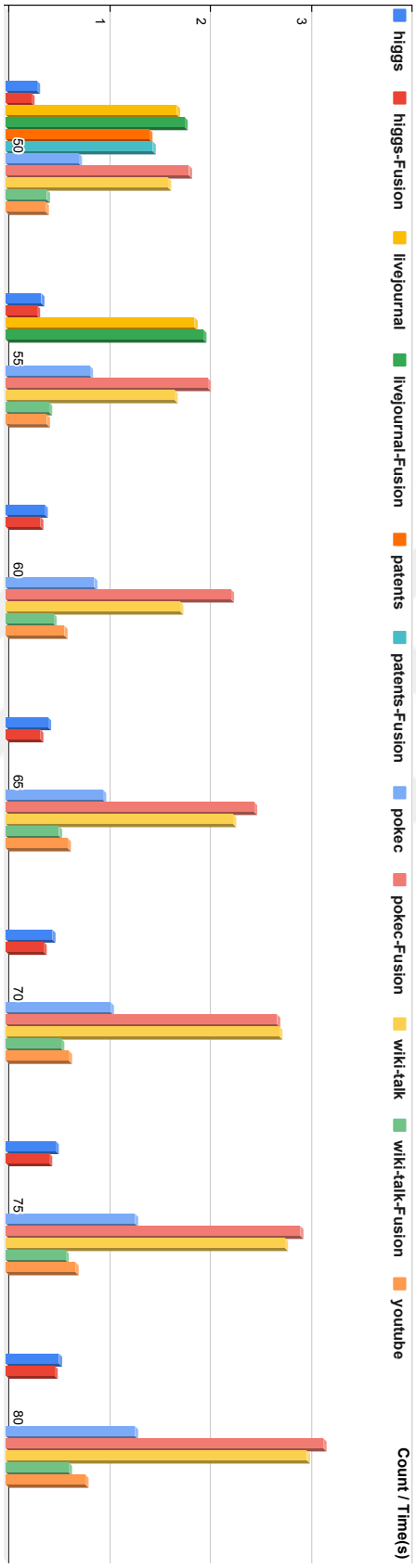


(d) Label Propagation

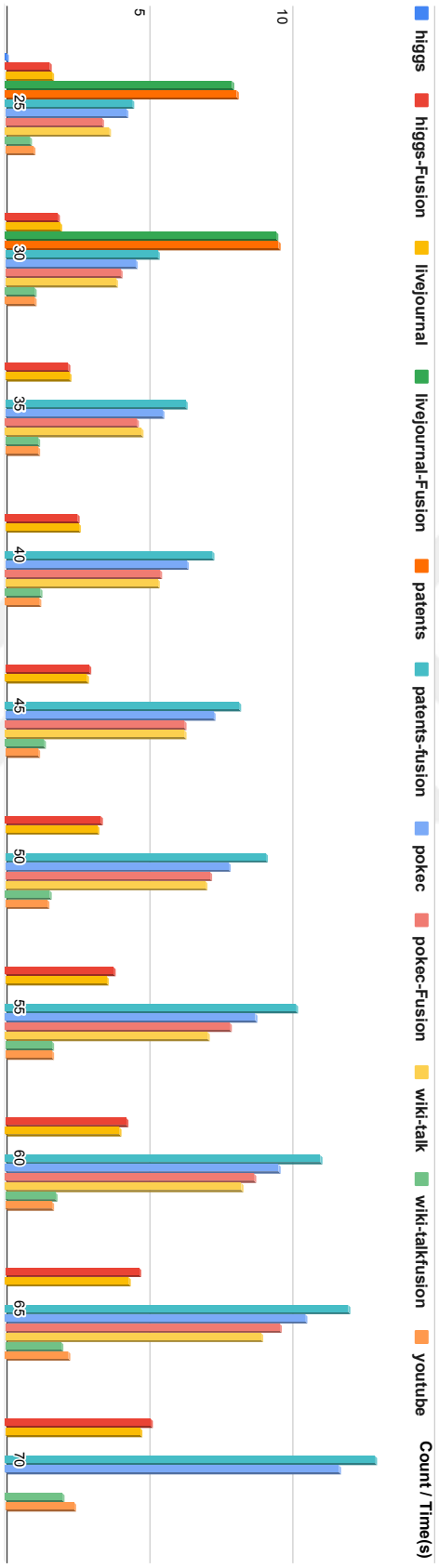
Figure 5.1: Homogeneous job set performance evaluation results on platform 1


1
2
40
45
50
55
60
65
70
higgs
higgs-Fusion
livejournal
livejournal-Fusion
patents
patents-Fusion
pokec
pokec-Fusion
wiki-talk
wiki-talk-Fusion
youtube
youtube-Fusion
Job Count / Time(s)

(a) BFS


1
2
3
60
55
60
65
70
75
80
higgs
higgs-Fusion
livejournal
livejournal-Fusion
patents
patents-Fusion
pokec
pokec-Fusion
wiki-talk
wiki-talk-Fusion
youtube
Count / Time(s)

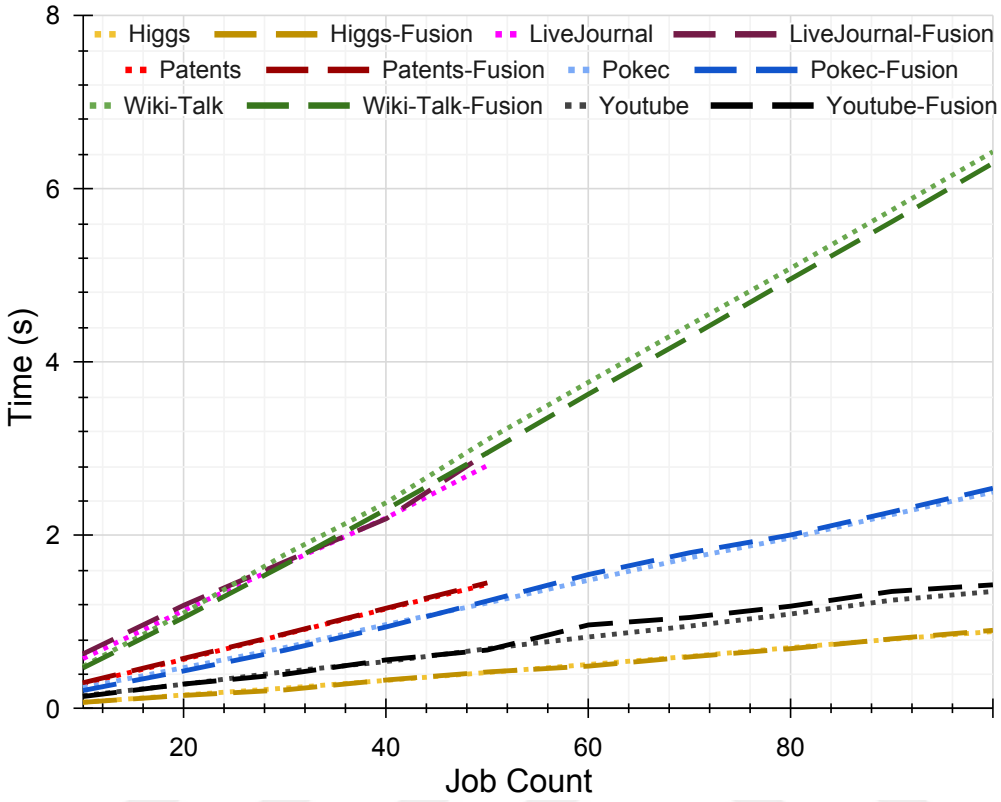
(b) SSSP


5
10
25
30
35
40
45
50
55
60
65
70
higgs
higgs-Fusion
livejournal
livejournal-Fusion
patents
patents-fusion
pokec
pokec-Fusion
wiki-talk
wiki-talkfusion
youtube
Count / Time(s)

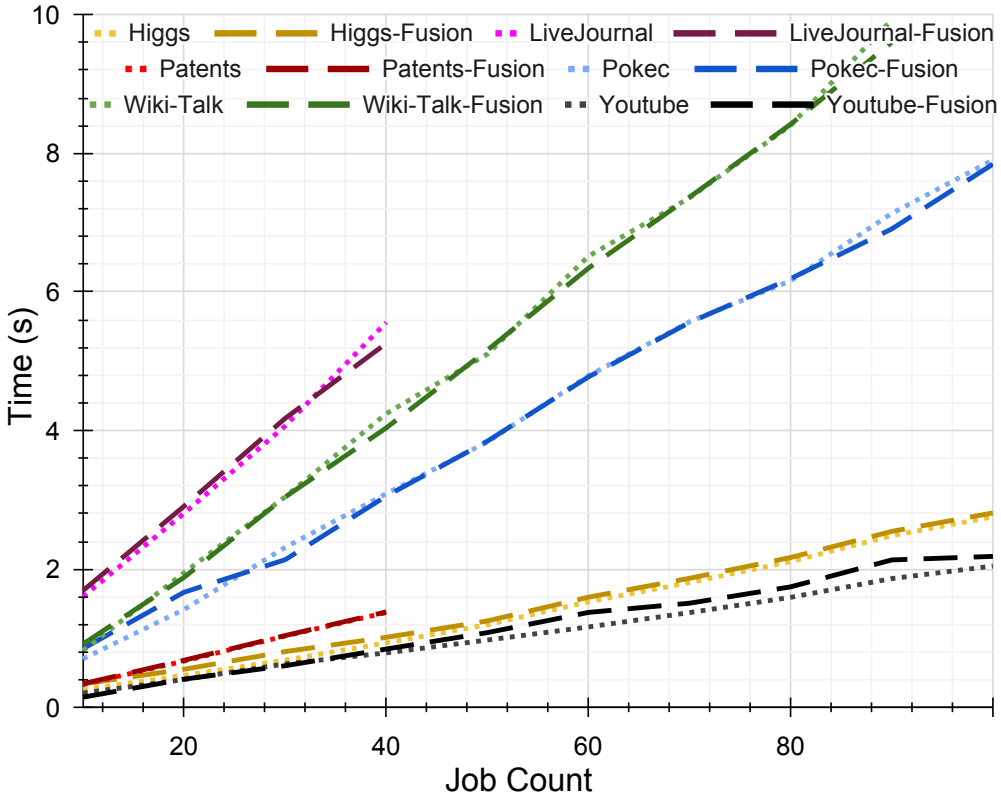
(c) PageRank

(d) Label Propagation

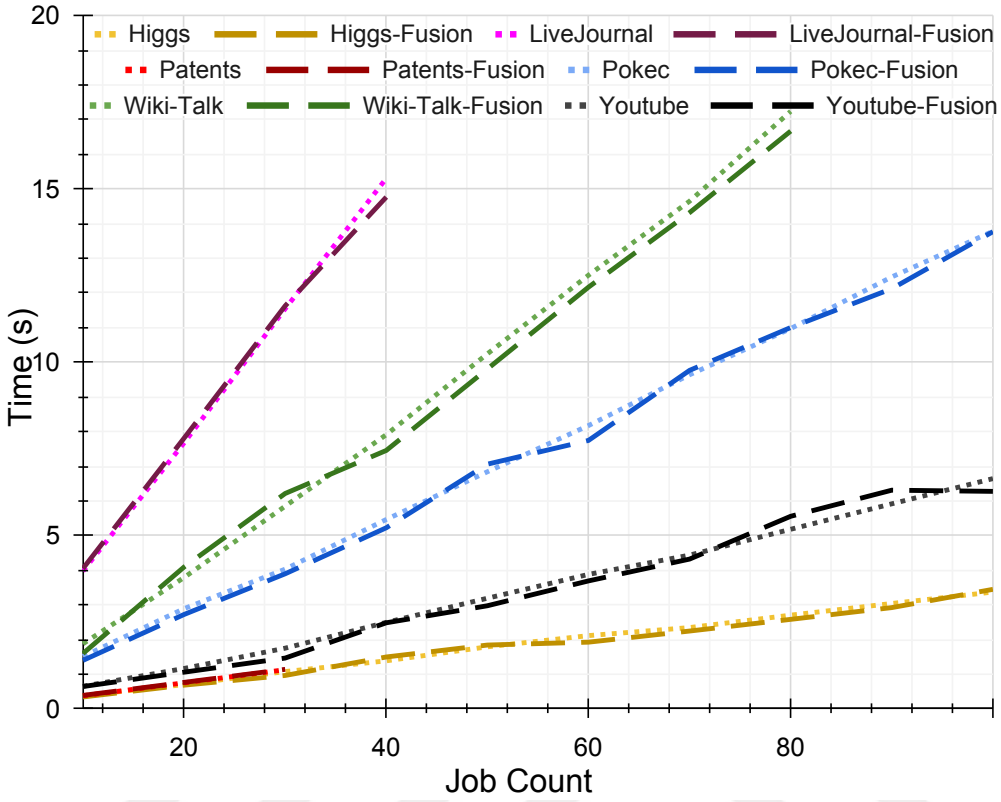
Figure 5.2: Homogeneous job set performance results on platform 1



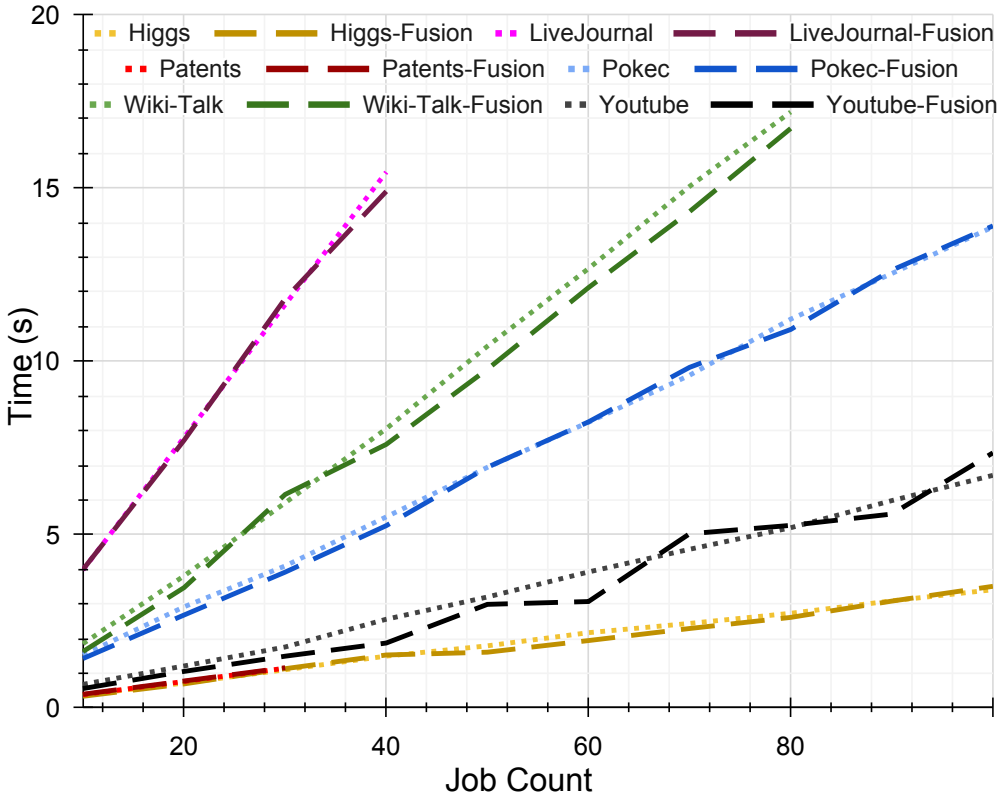
(a) Hetero 1



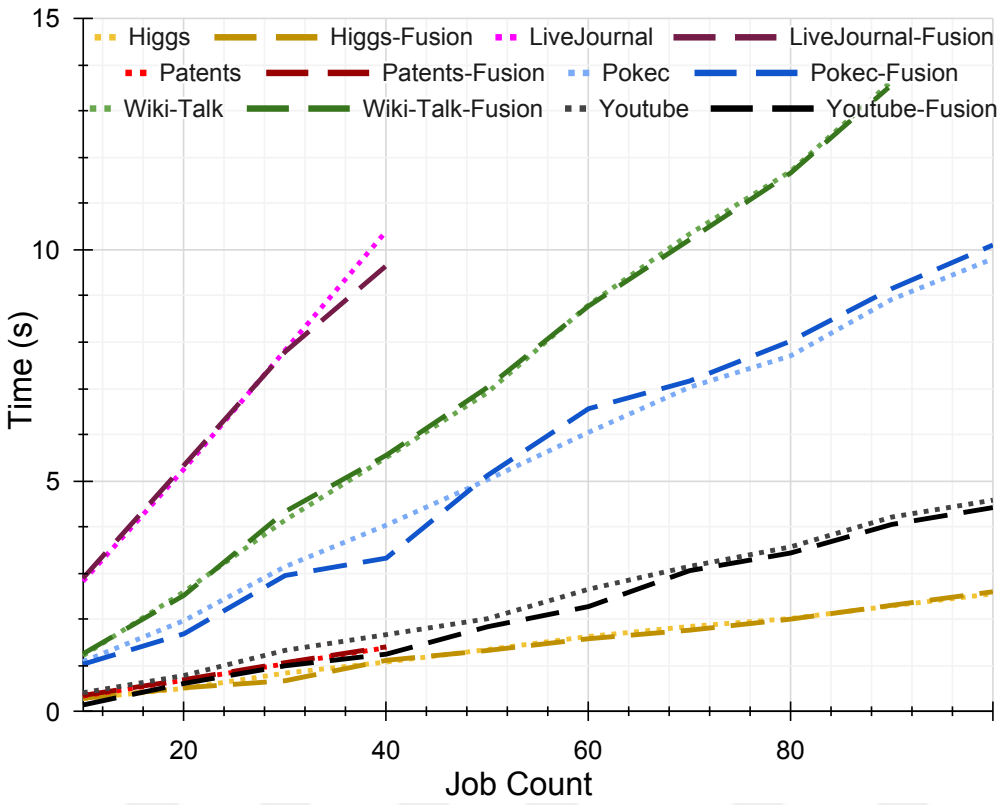
(b) Hetero 2



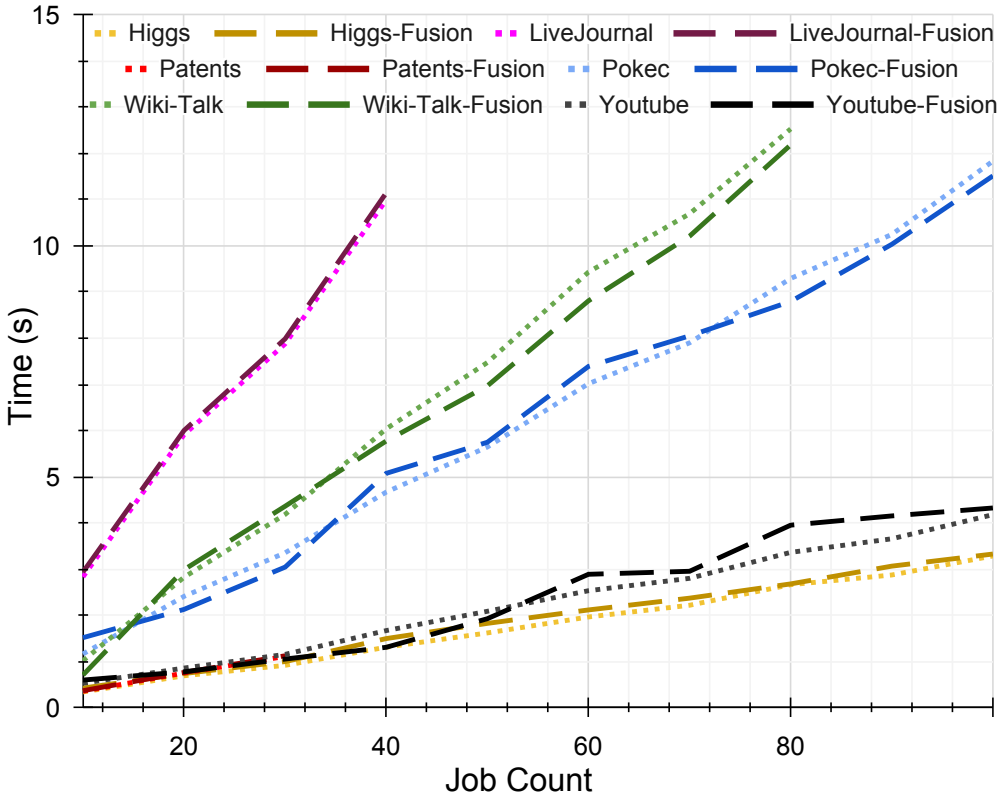
(c) Hetero 3



(d) Hetero 4

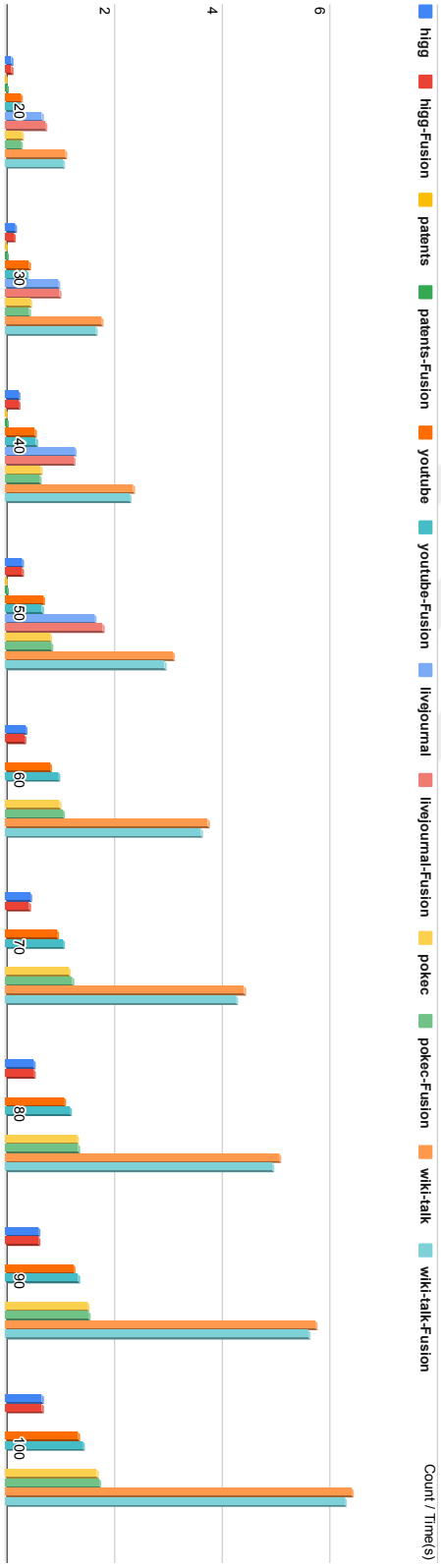


(e) Hetero 5

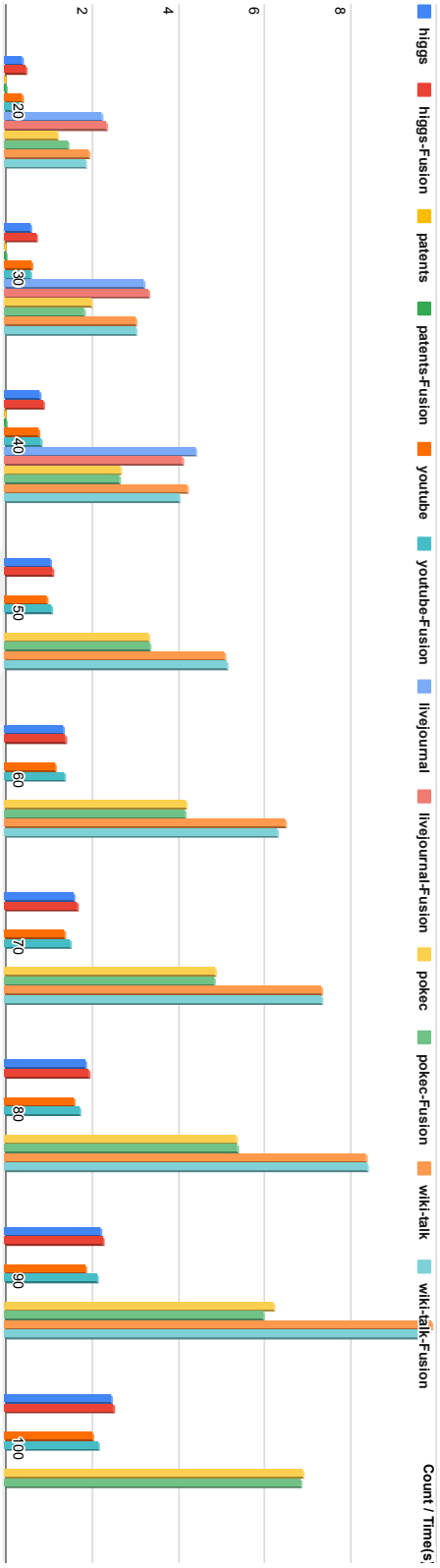


(f) Hetero 6

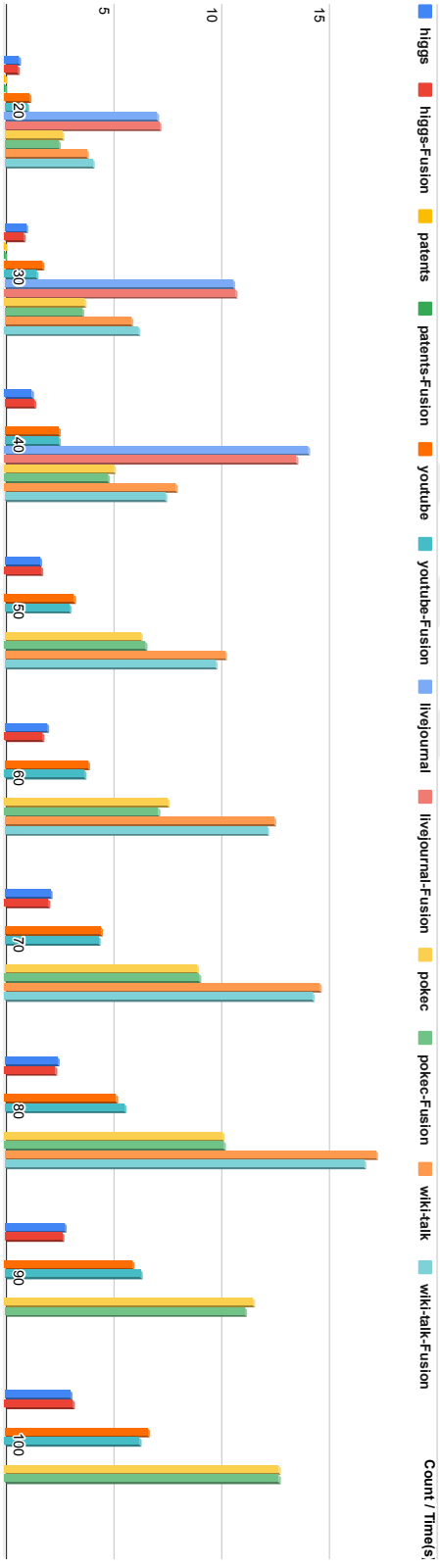
Figure 5.3: Heterogeneous job set performance evaluation results on platform 1



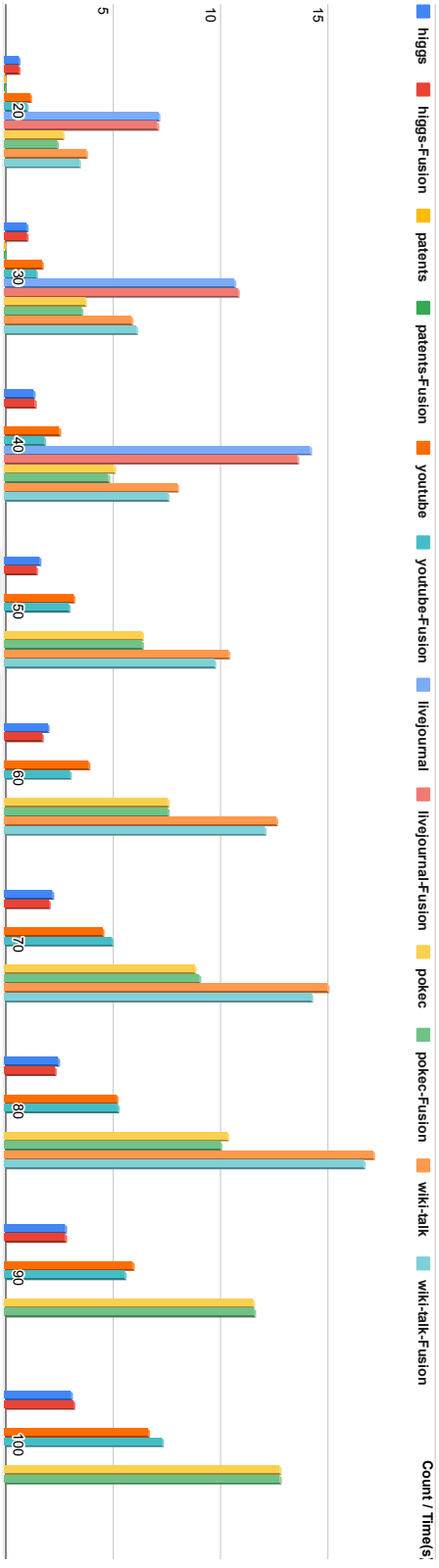
(a) Hetero 1



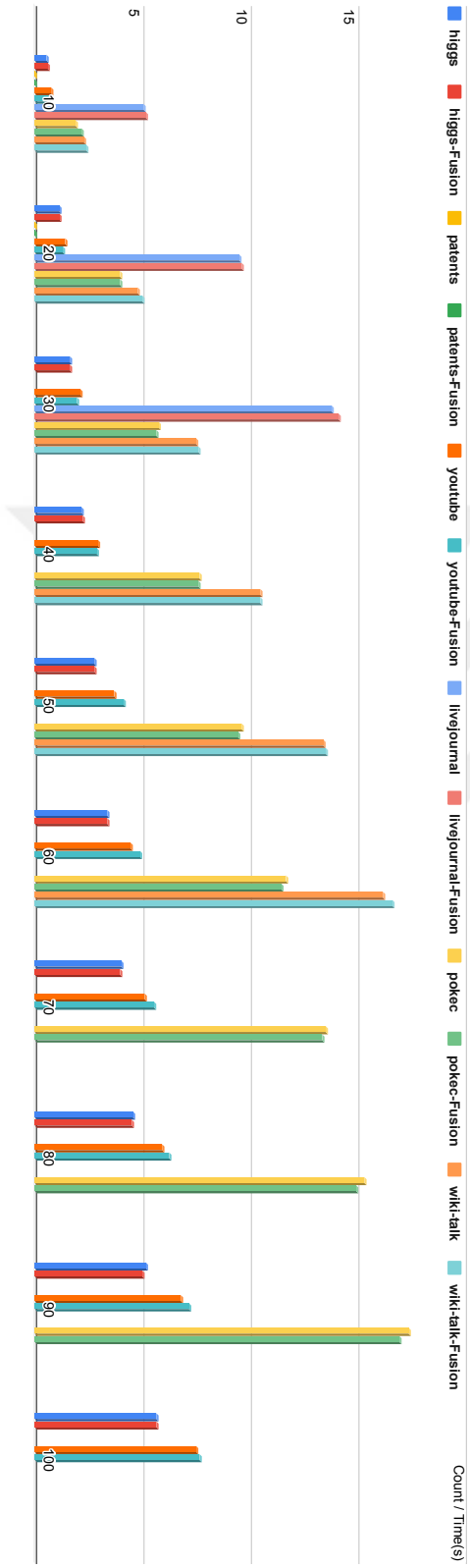
(b) Hetero 2



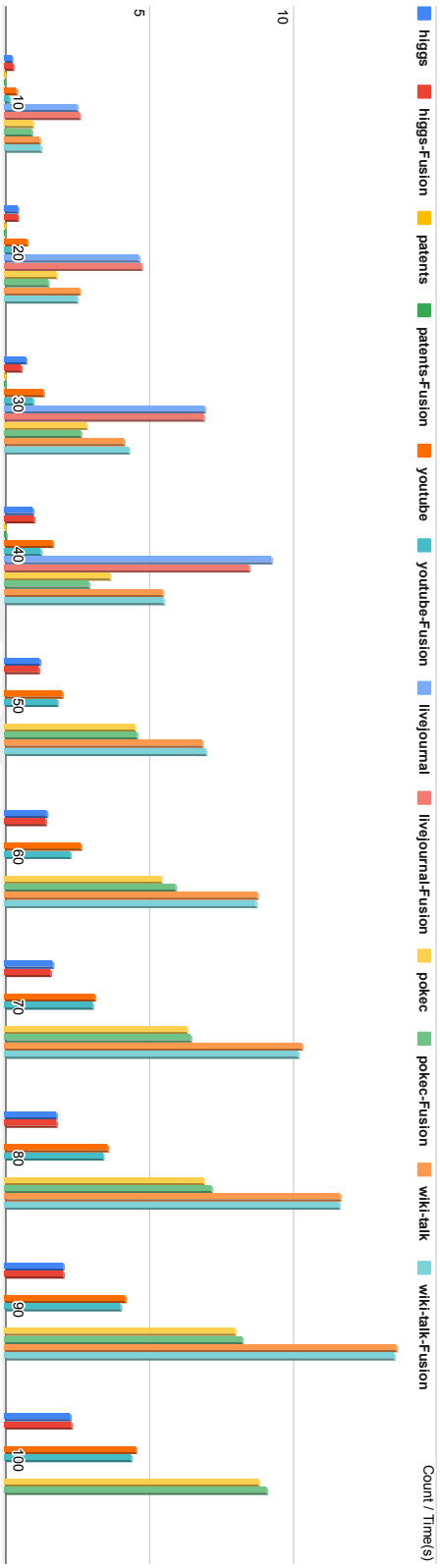
(c) Hetero 3



(d) Hetero 4

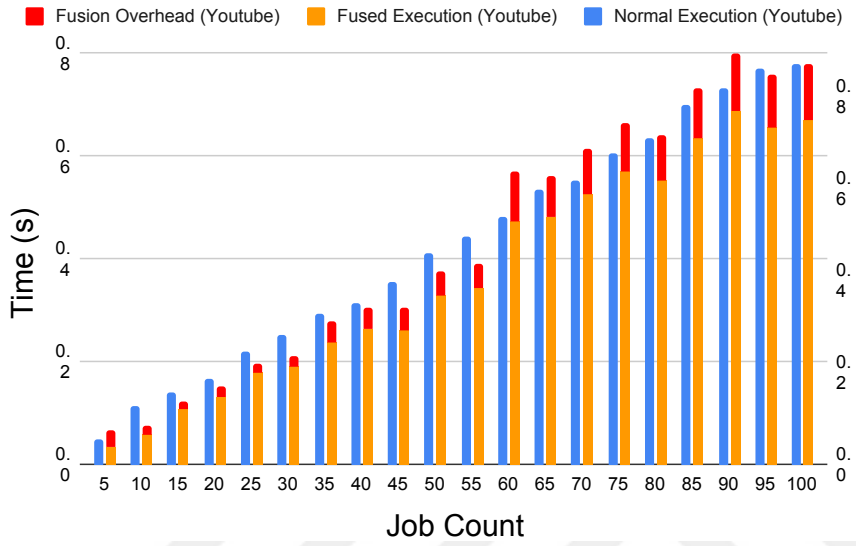


(e) Hetero 5

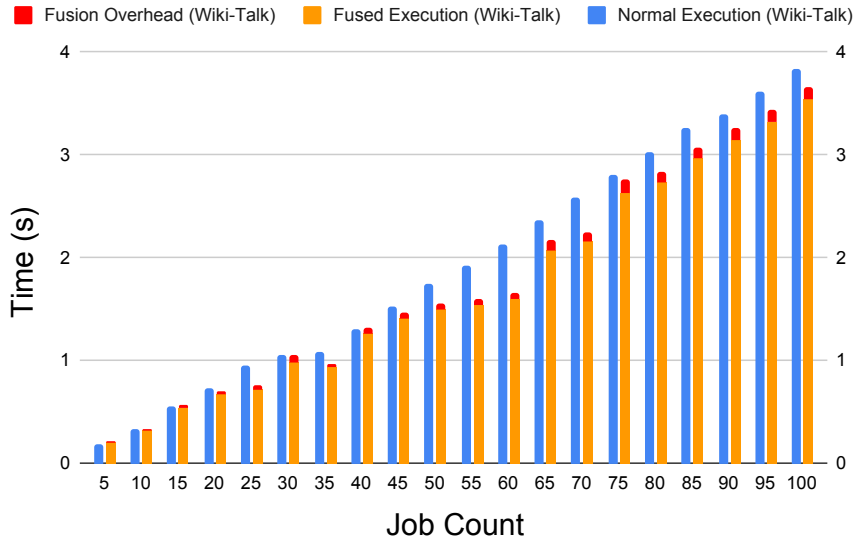


(f) Hetero 6

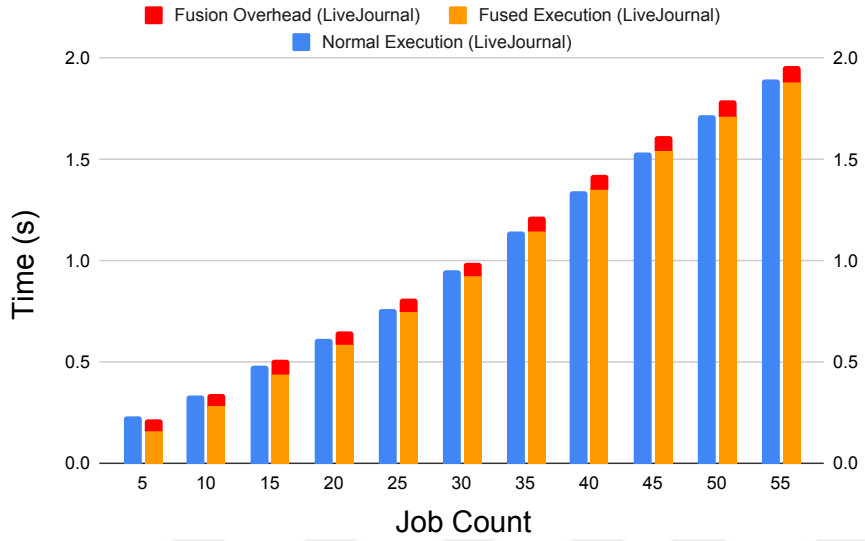
Figure 5.4: Heterogeneous job set performance results on platform 1



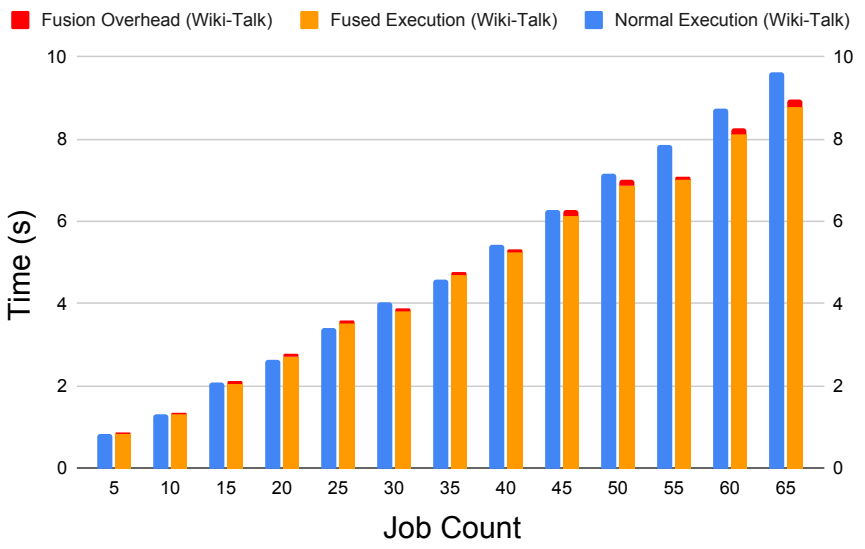
(a) BFS/Youtube (Platform 1)



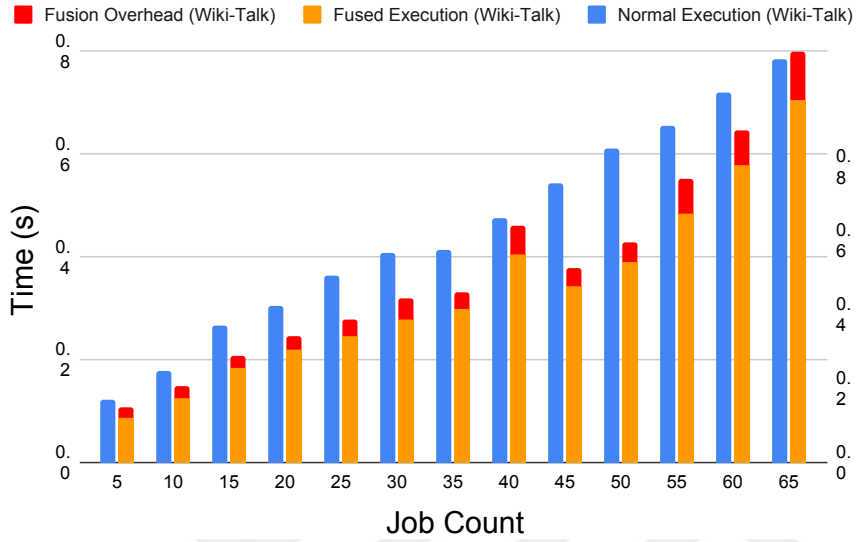
(b) BFS/Wiki-Talk (Platform 1)



(c) BFS/LiveJournal (Platform 1)



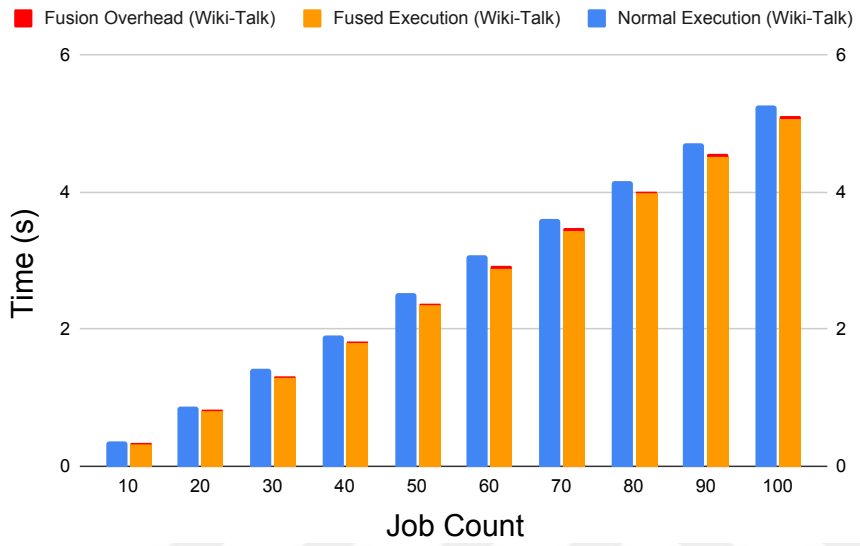
(d) PageRank/Wiki-Talk (Platform 1)



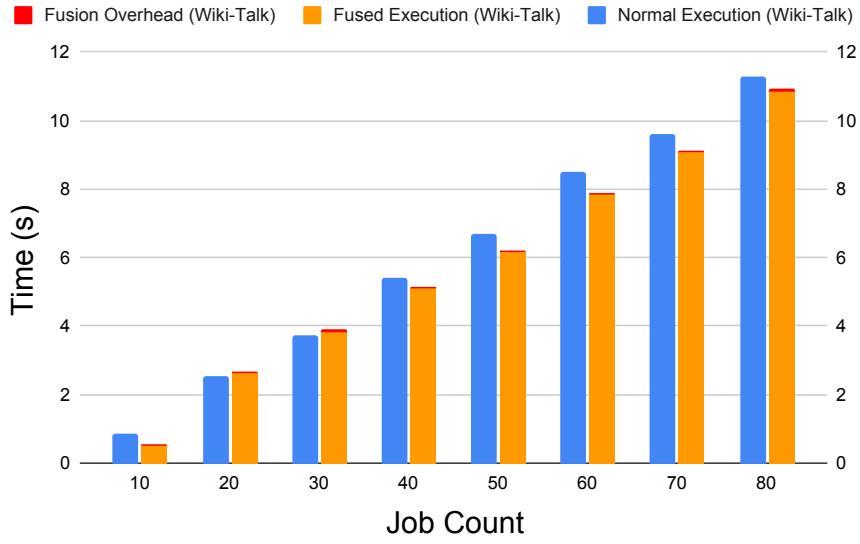
(e) LP/Wiki-Talk (Platform 1)

(f) SSSP/Wiki-Talk (Platform 2)

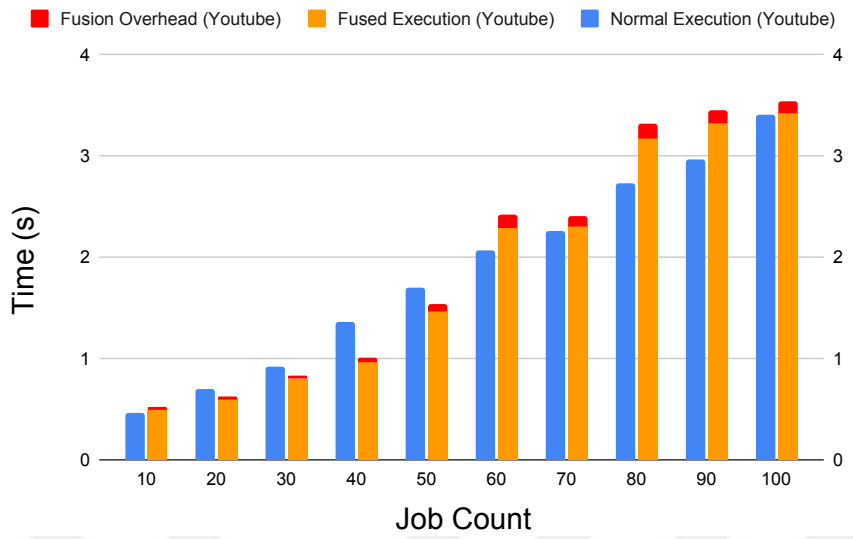
Figure 5.5: Fusion overhead on select homogeneous job sets and graphs



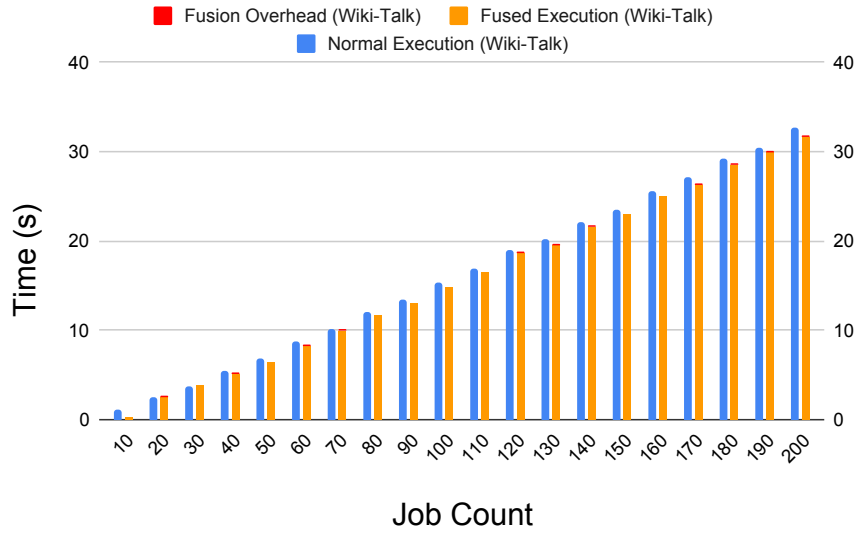
(a) Hetero 1/Wiki-Talk (Platform 1)



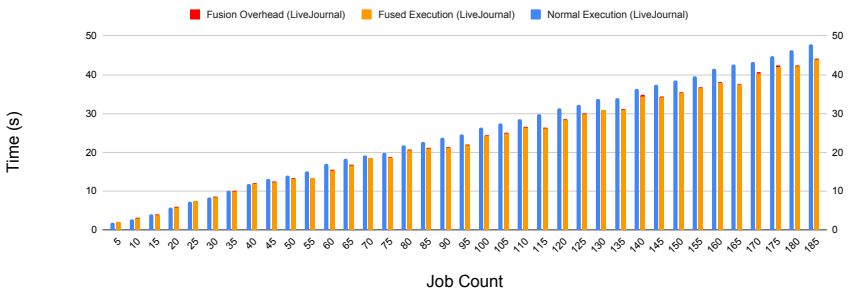
(b) Hetero 6/Wiki-Talk (Platform 1)



(c) Hetero 6/Youtube (Platform 1)

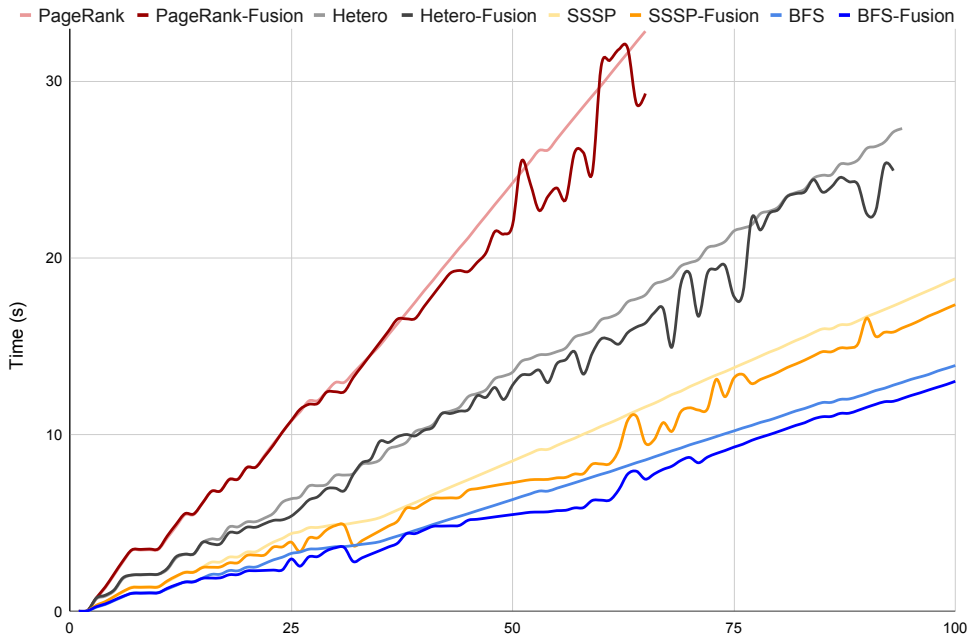


(d) Hetero 6/Wiki-Talk (Platform 3)

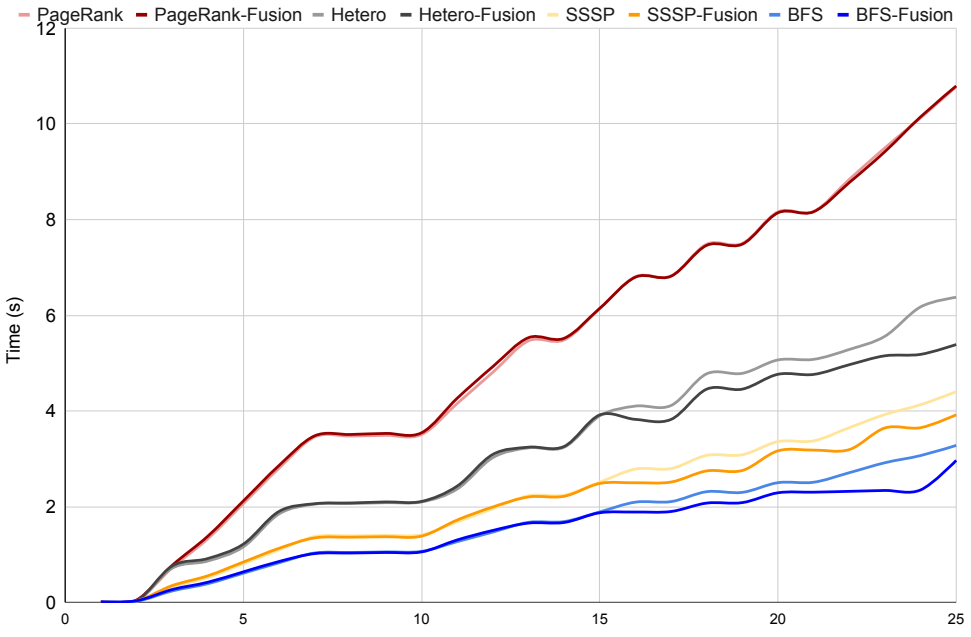


(e) Hetero 6/LiveJournal (Platform 2)

Figure 5.6: Fusion overhead on select heterogeneous job sets and graphs



(a) Runtime of executing multiple jobs (BFS, SSSP, PageRank) homogeneously and heterogeneously with and without kernel fusion, for 1 to 100 parallel jobs.



(b) Zoomed-in part of Figure 5.7a for 1 to 25 parallel jobs.

Figure 5.7: Runtime comparison of different homogeneous and heterogeneous jobs with and without fusion.

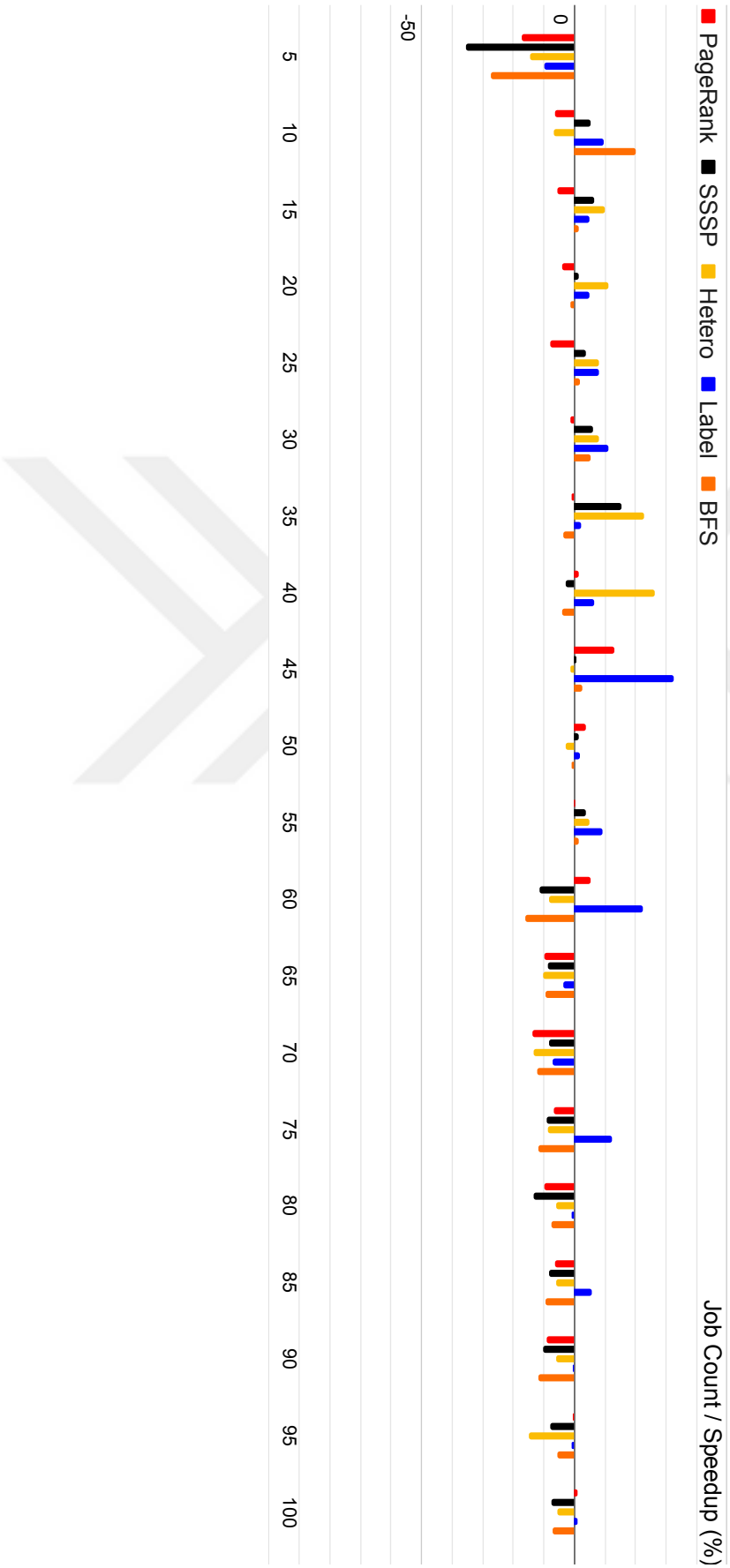


Figure 5.8: Speedup provided by kernel fusion for different job counts.

Chapter 6

CONCLUSION

Kernel fusion is an approach that fuses computation cores (kernels) in parallel jobs so that their performance can be optimized. In this work specifically, automated kernel fusion is used to fuse accesses to the same graph nodes and corresponding regions of memory by multiple parallel jobs in an automated fashion, so that memory caches are more efficiently utilized, resulting in performance improvements.

This dissertation presents a framework for automated GPU kernel fusion as well as extensive evaluation of the framework. The framework enables definition of GPU computational jobs in a specific manner, and then automatically fuses them at runtime by running jobs that access similar graph nodes to reduce memory access overhead.

The evaluations show that the framework is effective and yields about 5% to 10% speedup when a reasonably high number of jobs are executed in parallel.

6.1 Future Work

There are several areas of future research to achieve improvements for kernel fusion in GPUs.

First, the fusion algorithm can be modified to select jobs in manners that further reduces the memory footprint of each iteration. This work uses a relatively simple approach as a proof of concept to demonstrate effectiveness of memory access coherency in performance improvements. However, we are aware that other – potentially more accurate – shared frontier approaches can be utilized.

Second, instead of focusing on individual nodes as the shared frontier, fusion can work more directly on the regions of memory accessed by each job. In this work, it was assumed that graph nodes are a proxy for memory regions. This is

true in certain jobs such as BFS and SSSP, but not true in other jobs such as Label Propagation, that need to access more than each node and its direct edges. Considering the existence of a meta-compiler, more accurate data-flow analysis can be used to target direct memory accesses as the frontier, rather than the set of nodes under operation.

Thirdly, the amplification of parallel job numbers can be achieved through the implementation of advanced engineering techniques that specifically leverage contemporary GPU programming features. This involves facilitating the direct transfer of data streams from the CPU to the GPU for job execution, as opposed to the conventional practice of copying the entire job metadata to the GPU upon initialization. It is important to note that this particular approach was not adopted in the current research, as our focus was on prioritizing portability and robustness.

Fourth, dynamic job dispatch systems can be introduced that take new jobs and job definitions at runtime, and continuously dispatch them for parallel processing on the GPU. A dynamic dispatch system would result in a more significant impact of kernel fusion, as the ordering and throughput of jobs can be more finely controlled.

Fifth, kernel fusion can be more fine-grained, controlling exact nodes accessed by each job, in contrast with the current approach that only enables or disables individual jobs for each iteration. However, to make kernel fusion more fine-grained, job structures need to be redefined to provide more accurate control by the framework on individual nodes accessed by each job kernel.

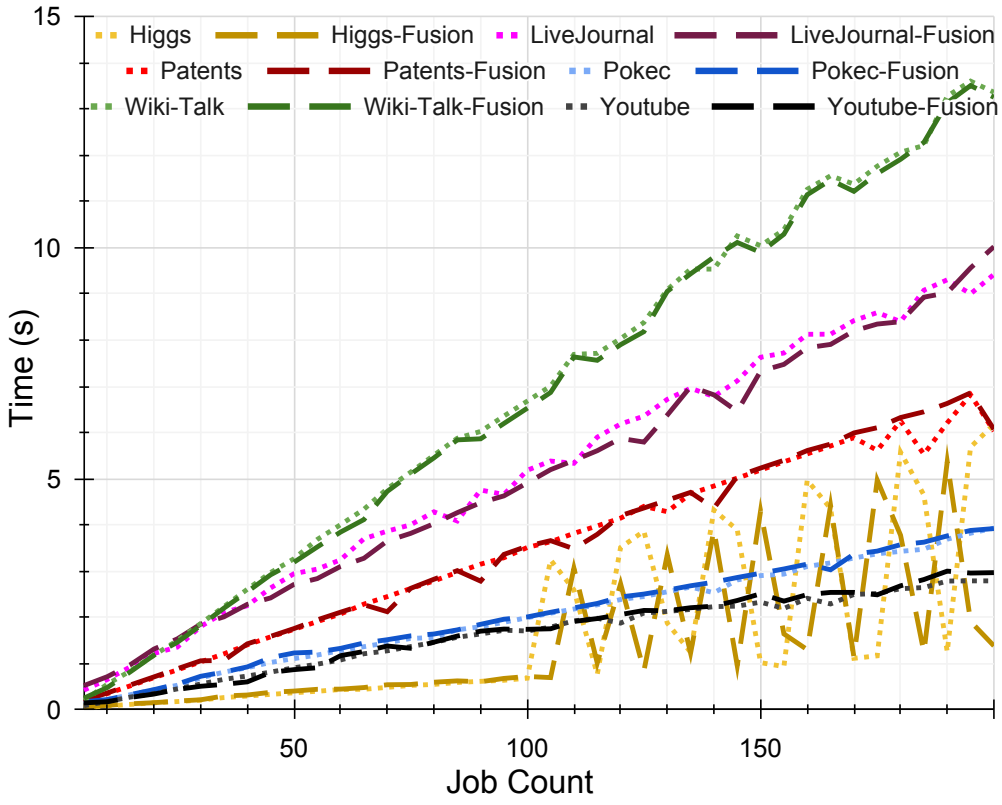
Finally, multiple GPUs and distributed systems are another avenue for expansion of automated kernel fusion in GPUs.

Appendix A

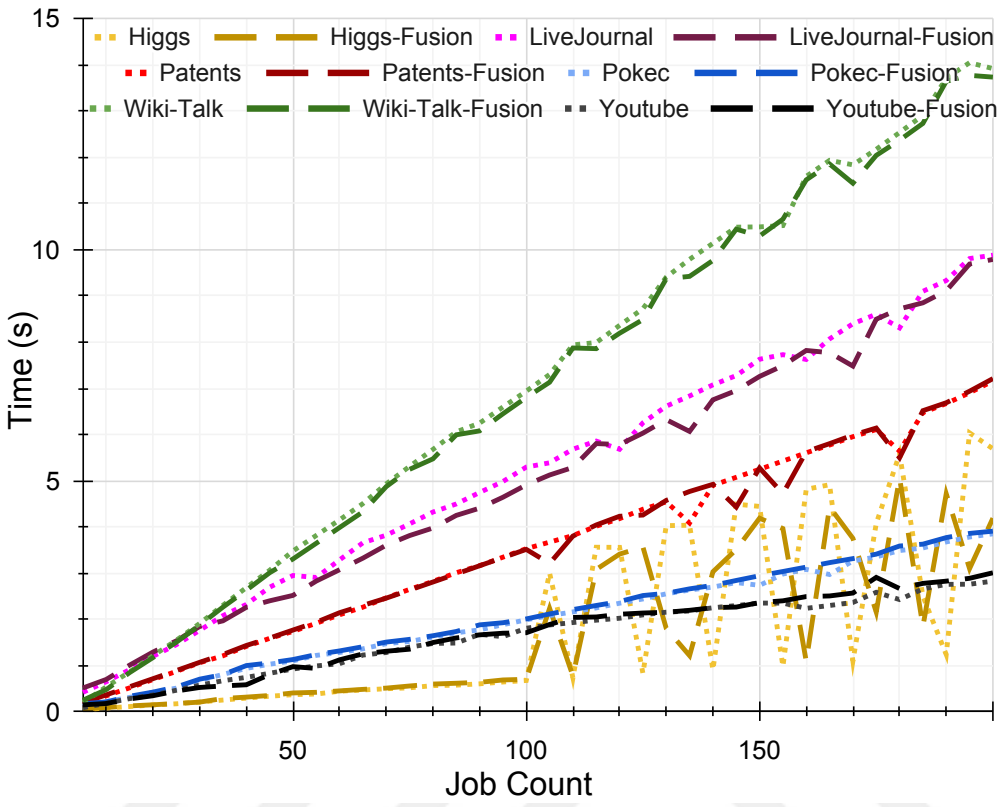
EVALUATION DIAGRAMS

In this appendix, additional evaluation diagrams relating to platforms 2 as defined in Table 5.4 are presented. These diagrams were omitted from the Section 5.2 for brevity.

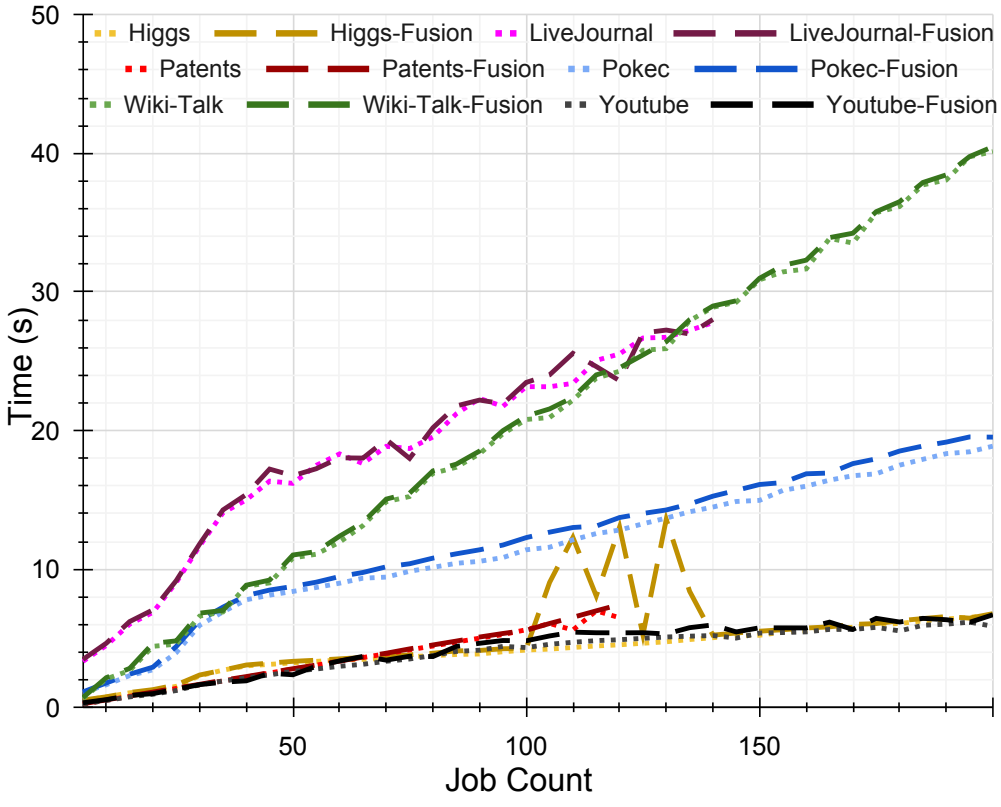
A.1 Platform 2



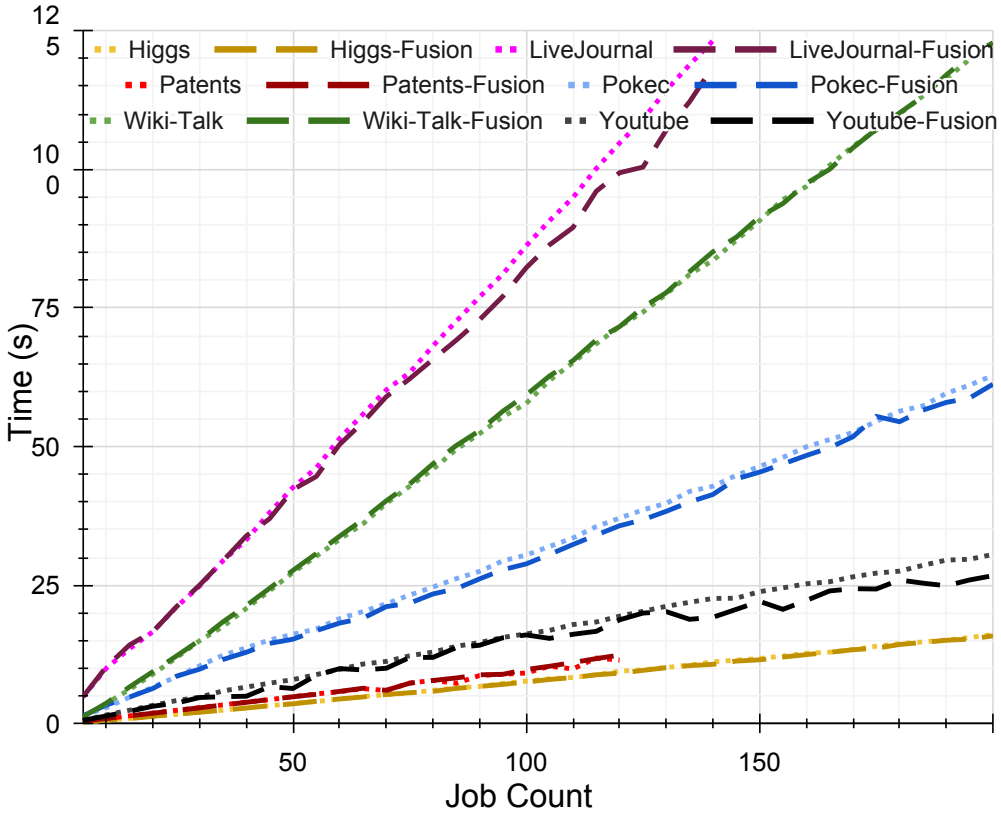
(a) BFS



(b) SSSP

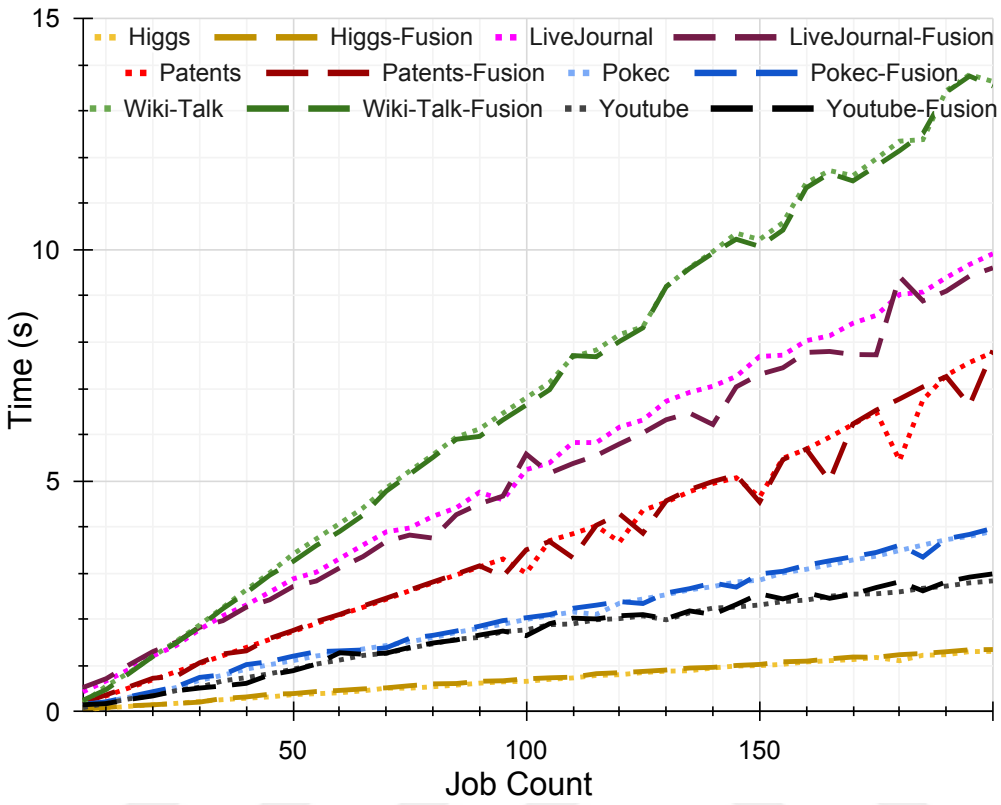


(c) PageRank

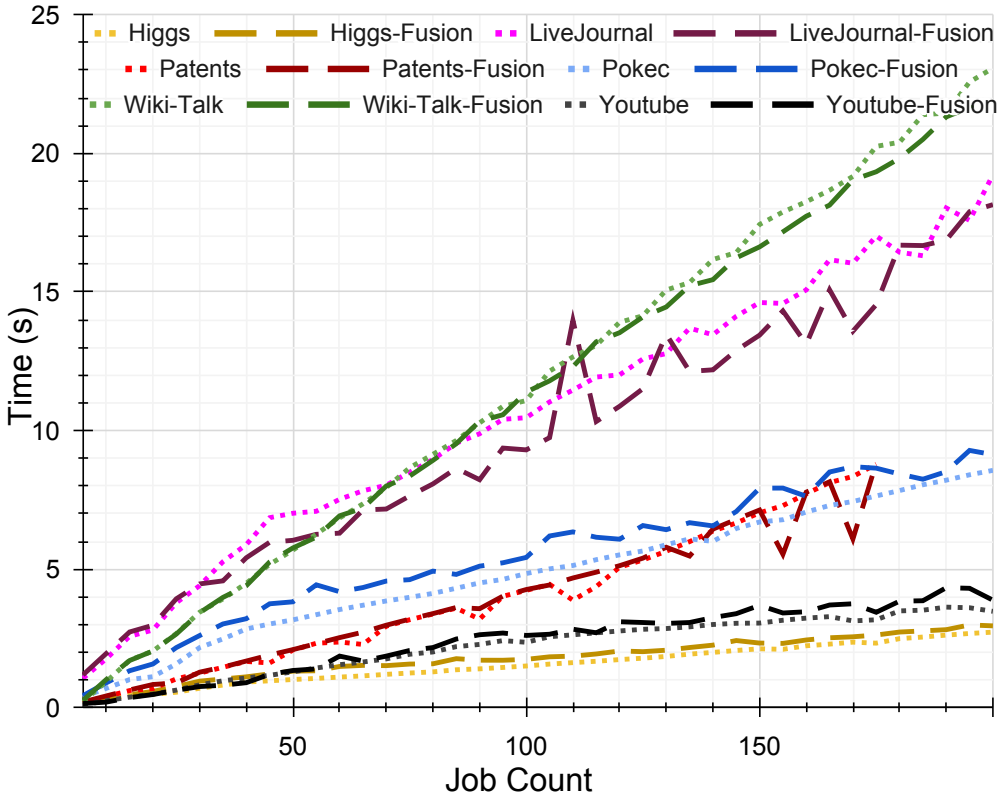


(d) Label Propagation

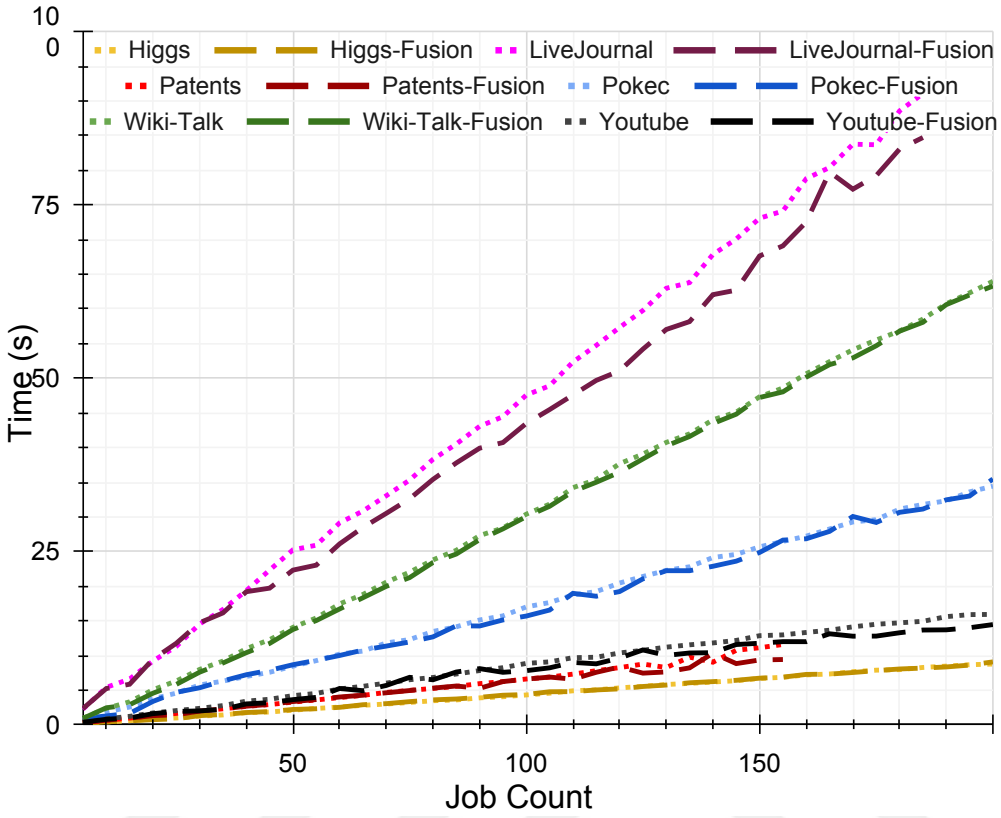
Figure A.1: Homogeneous job set performance evaluation results on platform 2



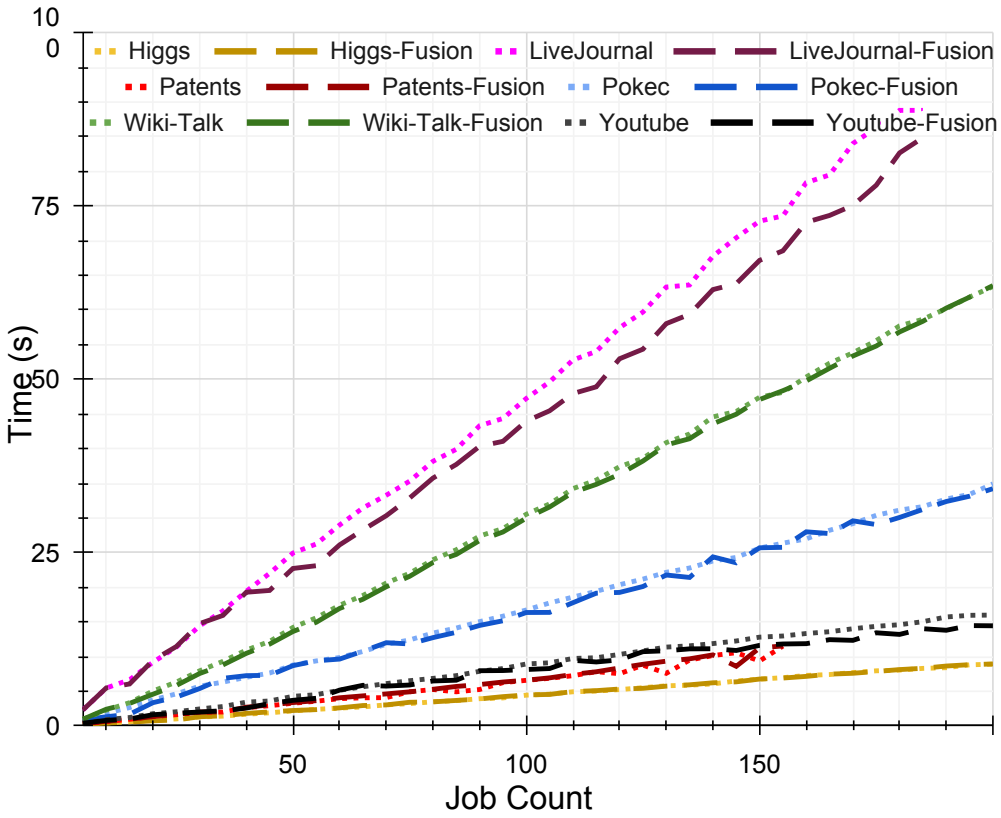
(a) Hetero 1



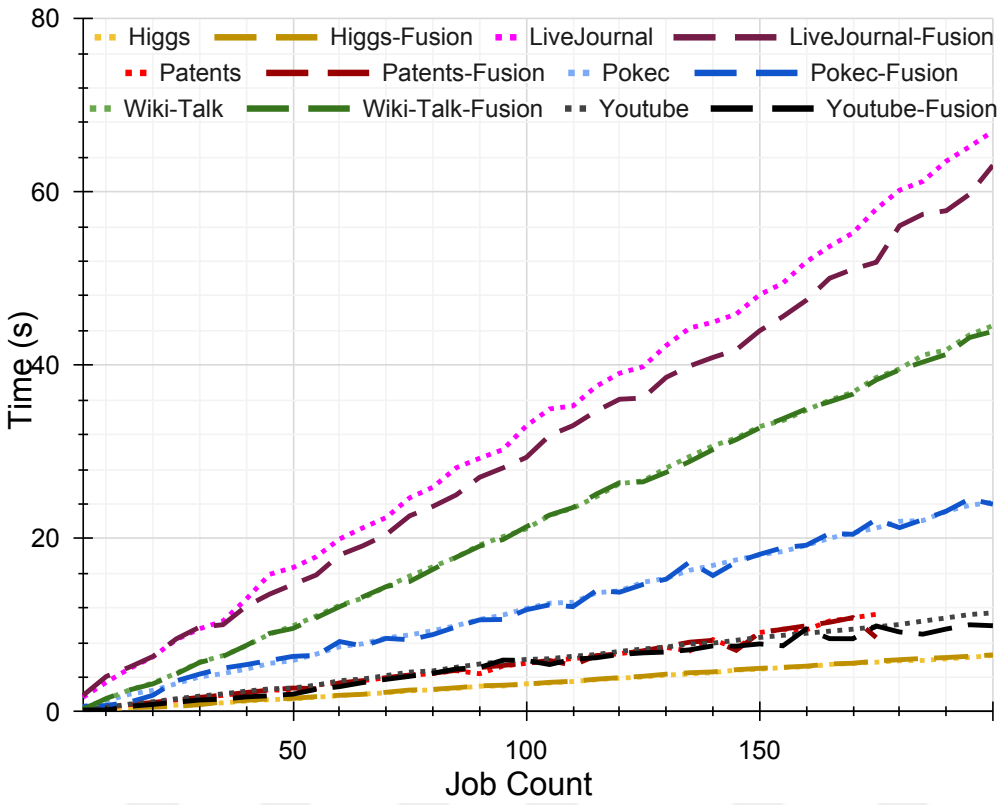
(b) Hetero 2



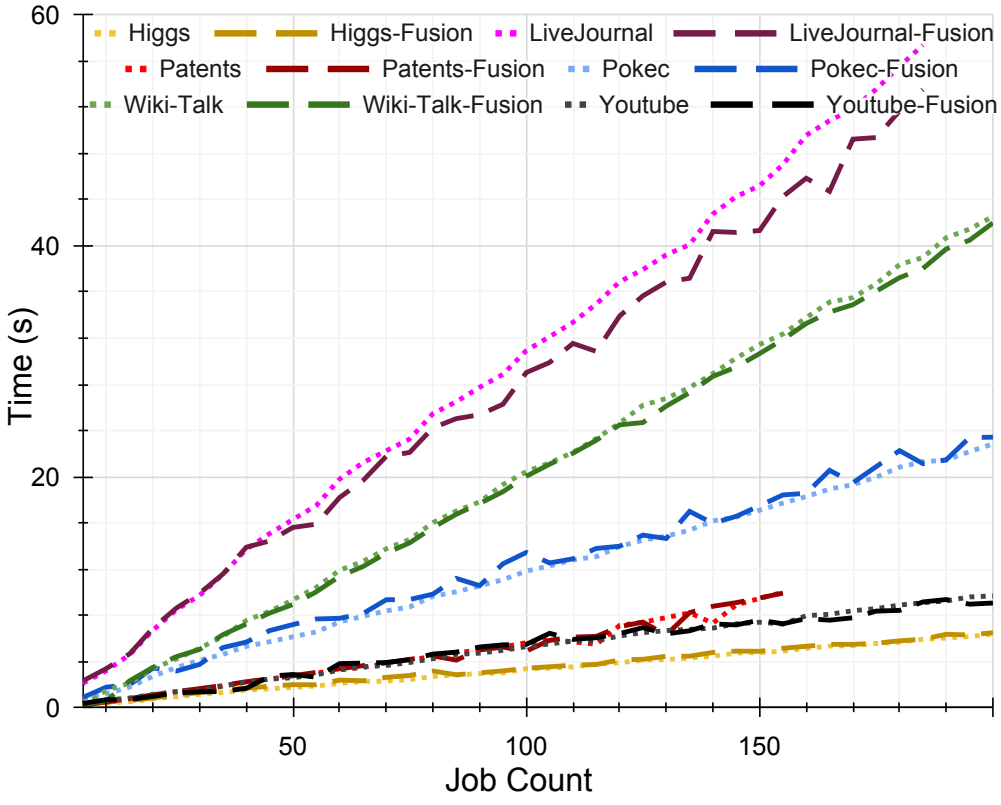
(c) Hetero 3



(d) Hetero 4



(e) Hetero 5



(f) Hetero 6

Figure A.2: Heterogeneous job set performance evaluation results on platform 2

BIBLIOGRAPHY

[Ben-Nun et al., 2020] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. (2020). Groute: Asynchronous multi-gpu programming model with applications to large-scale graph processing. *ACM Transactions on Parallel Computing (TOPC)*, 7(3):1–27.

[Chen et al., 2021] Chen, H., Shen, M., Xiao, N., and Lu, Y. (2021). Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16.

[Filipovič et al., 2015] Filipovič, J., Madzin, M., Fousek, J., and Matyska, L. (2015). Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957.

[Gharaibeh et al., 2013] Gharaibeh, A., Reza, T., Santos-Neto, E., Costa, L. B., Sallinen, S., and Ripeanu, M. (2013). Efficient large-scale graph processing on hybrid cpu and gpu systems. *arXiv preprint arXiv:1312.3018*.

[Hong et al., 2017] Hong, C., Sukumaran-Rajam, A., Kim, J., and Sadayappan, P. (2017). Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40. IEEE.

[Ismayilov et al., 2023] Ismayilov, I., Baydamirli, J., Sağbili, D., Wahib, M., and Unat, D. (2023). Multi-gpu communication schemes for iterative solvers: When cpus are not in charge. In *Proceedings of the 37th International Conference on Supercomputing*, pages 192–202.

[Khorasani et al., 2014] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. (2014). Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252.

[Leskovec and Krevl, 2014] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[Low et al., 2014] Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J. (2014). Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146.

[Mazloumi et al., 2019] Mazloumi, A., Jiang, X., and Gupta, R. (2019). Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 349–358. IEEE.

[Nvidia Corporation, 2022a] Nvidia Corporation (2022a). Cuda.

[Nvidia Corporation, 2022b] Nvidia Corporation (2022b). Nvidia cuda compiler driver nvcc.

[Nvidia Corporation, 2022c] Nvidia Corporation (2022c). Nvidia driver 10.47.03.

[Pan and Li, 2017] Pan, P. and Li, C. (2017). Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 217–224. IEEE.

[Park et al., 2022] Park, J., Shim, Y., Lee, F., Rammohan, A., Goyal, S., Shim, M., Jeong, C., and Kim, D. S. (2022). Prediction and interpretation of polymer

properties using the graph convolutional network. *ACS Polymers Au*, 2(4):213–222.

[Shi et al., 2017] Shi, X., Luo, X., Liang, J., Zhao, P., Di, S., He, B., and Jin, H. (2017). Frog: Asynchronous graph processing on gpu with hybrid coloring model. *IEEE Transactions on Knowledge and Data Engineering*, 30(1):29–42.

[Shi et al., 2018] Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B., and Hua, Q.-S. (2018). Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35.

[Shun and Blelloch, 2013] Shun, J. and Blelloch, G. E. (2013). Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146.

[Siek et al., 2001] Siek, J. G., Lee, L.-Q., and Lumsdaine, A. (2001). *The Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education.

[Strohmaier, 2006] Strohmaier, E. (2006). Top500 supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 18–es, New York, NY, USA. Association for Computing Machinery.

[Sun et al., 2017] Sun, J., Vandierendonck, H., and Nikolopoulos, D. S. (2017). Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, pages 1–10.

[Sundaram et al., 2015] Sundaram, N., Satish, N. R., Patwary, M. M. A., Dulloor, S. R., Vadlamudi, S. G., Das, D., and Dubey, P. (2015). Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*.

[Wahib and Maruyama, 2014] Wahib, M. and Maruyama, N. (2014). Scalable kernel fusion for memory-bound gpu applications. In *SC'14: Proceedings of the Inter-*

national Conference for High Performance Computing, Networking, Storage and Analysis, pages 191–202. IEEE.

[Wang et al., 2010] Wang, G., Lin, Y., and Yi, W. (2010). Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE.

[Wang et al., 2016] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2016). Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12.

[Wei et al., 2016] Wei, H., Yu, J. X., Lu, C., and Lin, X. (2016). Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828.

[Weinberg, 1976] Weinberg, S. (1976). Mass of the higgs boson. *Phys. Rev. Lett.*, 36:294–296.

[Xue et al., 2014] Xue, J., Yang, Z., Qu, Z., Hou, S., and Dai, Y. (2014). Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 227–238.

[Yang et al., 2019] Yang, C., Buluç, A., and Owens, J. D. (2019). Graphblast: A high-performance linear algebra-based graph framework on the gpu. corr abs/1908.01407 (2019). *arXiv preprint arXiv:1908.01407*.

[Zhang et al., 2016] Zhang, Q., Yan, D., and Cheng, J. (2016). Quegel: A general-purpose system for querying big graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2189–2192.

[Zhang et al., 2022] Zhang, Y., Liang, Y., Zhao, J., Mao, F., Gu, L., Liao, X., Jin, H., Liu, H., Guo, S., Zeng, Y., et al. (2022). Egraph: efficient concurrent gpu-based dynamic graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5823–5836.

[Zhang et al., 2018a] Zhang, Y., Liao, X., Jin, H., Gu, L., He, L., He, B., and Liu, H. (2018a). {CGraph}: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452.

[Zhang et al., 2018b] Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S. (2018b). Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30.

[Zhao et al., 2023] Zhao, J., Zhang, Y., Cheng, J., Wu, Y., Ye, C., Yu, H., Huang, Z., Jin, H., Liao, X., Gu, L., et al. (2023). Sagraph: A similarity-aware hardware accelerator for temporal graph processing. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

[Zhao et al., 2019] Zhao, J., Zhang, Y., Liao, X., He, L., He, B., Jin, H., Liu, H., and Chen, Y. (2019). Graphm: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14.

[Zhu et al., 2016] Zhu, X., Chen, W., Zheng, W., and Ma, X. (2016). Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316.