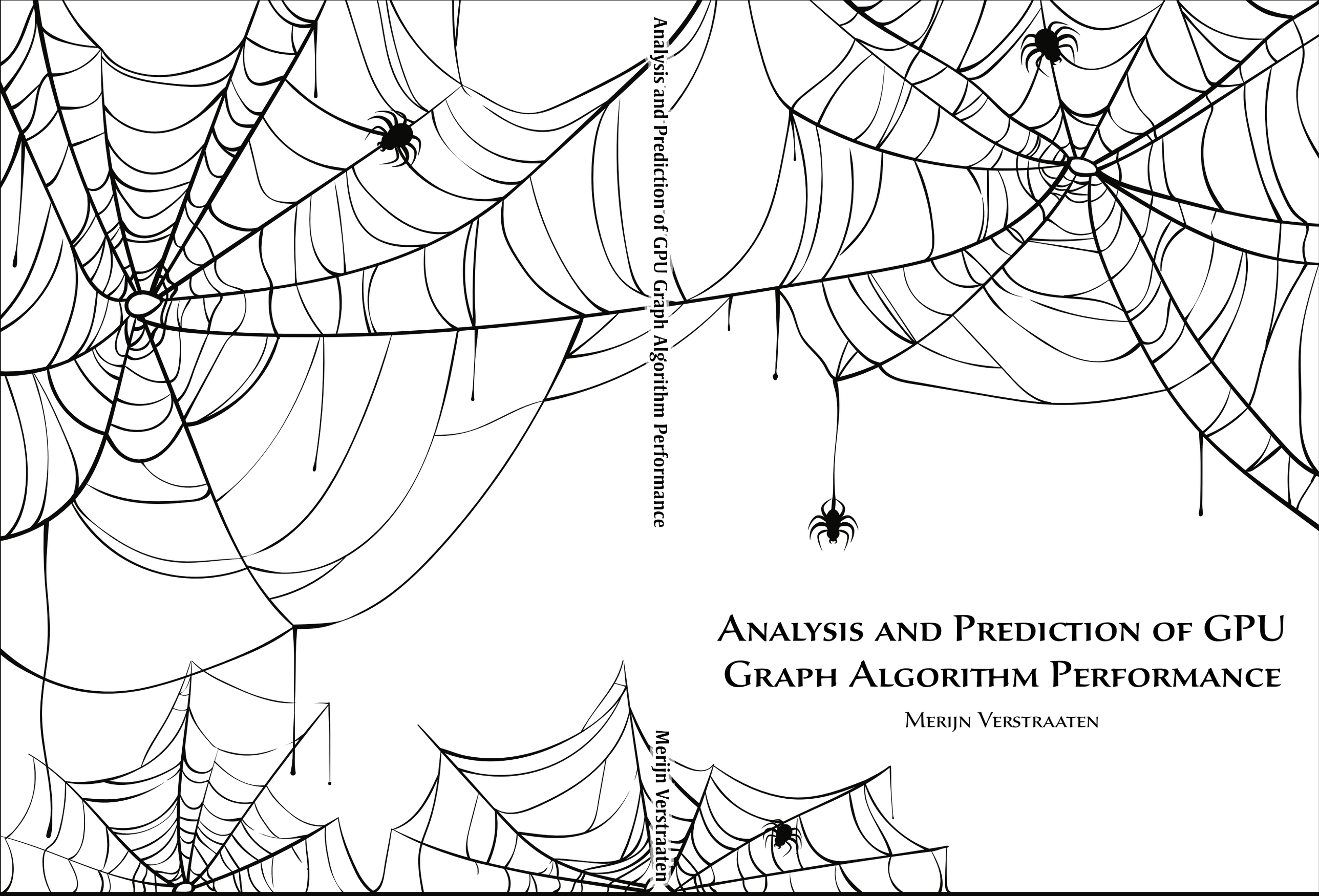


Analysis and Prediction of GPU Graph Algorithm Performance

Merijn Verstraaten

Analysis and Prediction of GPU Graph Algorithm Performance

Merijn Verstraaten

Analysis and Prediction of GPU Graph Algorithm Performance

Merijn Verstraaten

This research has been partially supported by the NWO Veni project *Graphitti: A Framework for Self-adapting High Performance Massively Parallel Graph Processing* (STW, dossiernummer 12480, 2012). This research has also been supported by the Distributed ASCI Supercomputer 5 (DAS-5).

Cover based on art by Zaie/Shutterstock.com
Images used under license from Shutterstock.com

Typeset using XeLaTeX, written using Vim.

ISBN:978-94-6421-837-4

Analysis and Prediction of GPU Graph Algorithm Performance

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. G.T.M. ten Dam
ten overstaan van een door het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel op
donderdag 8 september 2022, te 13:00 uur

door

Merijn Elwin Verstraaten

geboren te Amstelveen

Promotie Commissie

Promotor:	Prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
Copromotor:	dr. ir. A.L. Varbanescu	Universiteit van Amsterdam
Overige Leden:	prof. dr. J.D. Owens	University of California, Davis
	prof. dr. S. Scholz	Herriot-Watt University
	prof. dr. G.H.L. Fletcher	Technische Universiteit Eindhoven
	prof. dr. P.T. Groth	Universiteit van Amsterdam
	prof. dr. A.D. Pimentel	Universiteit van Amsterdam
	prof. dr. R.V. van Nieuwpoort	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

List of Figures

List of Tables

Preface & Acknowledgements

As I write these final[1] words, several thesis related deadlines are fast approaching. Some things never change. Out of the many things I learned during my Ph.D., "writing without a deadline looming over me like a Sword of Damocles" sadly wasn't one. As the saying goes: *de laatste loodjes wegen het zwaarst…*

By now I spent as much time working on this thesis in my own time as I spent employed at the university. I suspect the only reason I got to this point is that most valuable attribute for any Ph.D. — stubbornness. Ironically, this whole endeavour would have been completed a whole lot more efficiently had I not stubbornly decided to be "less stubborn" at the start.

> But you didn't get this far by giving up, did you? That's right. You have something called 'determination'.
>
> Undertale

This whole journey started in 2010[2]. I decided that I could not consider my Computer Science master complete without learning more about compilers. The Vrije Universiteit (VU) no longer had any compiler course, so I enrolled in Clemens' compiler construction course at the Universiteit van Amsterdam (UvA). In hindsight, one of the most useful and enlightening courses of my entire degree.

At the end of this course I approached Clemens to see if I could do a master's project with him on "something with programming languages". This ended up as a project to port S-Net to Intel's experimental Single-chip Cloud Computer (SCC), a thoroughly weird and experimental piece of hardware. Culminating in my first two "proper" scientific publications.

[1] Yet also first…

[2] Probably? Who knows after 12 years…

After my project finished and I finally graduated, I stayed in the Computer Systems Architecture (CSA) group as research assistant. During this period I met a lot of friends and colleagues who would show me how modern science (dis)functions. Starting, of course, with Clemens who is responsible for getting me there in the first place. Roy and Roeland, my fellow master students, who were perhaps wiser in not seeing things through to the very end. Michiel, who supervised my master project. Raphael, who was always ready to help with my technical problems and to derail productivity for hours through philosophical discussions. And many more: Roberta, Mike, Mark, Peter, and Simon.

I worked on the ADVANCE project, where I also had some great colleagues and verbal sparring partners. Daniel and Frank, my day-to-day collaborator on the S-Net codebase. Philip and Jan, always willing to ignore the actual project to talk about the philosophy of programming languages and functional programming. And Bodo, who is probably indirectly responsible for my love of whisky.

When the UvA's funding within ADVANCE ran out, I finished the remainder of the project with Bodo in Edinburgh. I had a great time there, but in the end I missed my social life in the Netherlands too much to move to Edinburgh for a longer period. Somewhere out there, there is an alternative universe where I stayed to do a Ph.D. in "SaC world domination", and sometimes I wonder what that universe looks like…

> Coming back to where you started is not the same as never leaving.
>
> Terry Pratchett,
> *A Hat Full of Sky*

After my time in Edinburgh, I came back to the Netherlands, the UvA, and most of the same colleagues. At this point, the old CSA group was no more; absorbed into the System and Network Engineering (SNE) group. Which became my new home, a lot of old faces, but also many new ones. Giulio, my fellow starting Ph.D. student. Pieter, who was always up for discussions about computer science education and programming languages. My fellow "beer researchers": Ralph, Cosmin, Stijn, Taddeüs, Reggie, Łukasz, Mikołaj, Karel, and Joe. And alsp Fahimeh, Ameneh, Paola, and everyone else I'm probably forgetting.

And most importantly Ana. Somewhere in late 2013 Raphael put me in contact with Ana. I was looking for a Ph.D. position, she was looking for a Ph.D. student, and Raphael thought we would get along. Correctly, it turns out. Our first conversation boiled down to the question: "Would you like to work on graph processing on Graphical Processing Units (GPUs)?"

I enjoyed the challenge of squeezing the maximum performance out of GPUs during my master's degree, so this seemed like an interesting challenge. I wasn't sure how hard it would be to do something new; after all, we've been doing graph theory for over 280 years [31], General Processing on GPU (GPGPU) programming for over 17 years [53], and even graph processing on GPUs for over 14 years [67]. Surely, the most obvious problems were already tackled by other researchers…right?

> This is the point in your career where you look around for the cavalry and realize that you're it…
>
> Edward Kmett

It turns out that *that* was hopelessly naive of me. What started out as lofty and ambitious plans of linking the structure of graphs to the best performing parallelisation strategy quickly ran into multiple walls. No one had really figured out a *necessary and sufficient* description of graph structure. Sure, there are several tens (if not hundreds) of different structural properties in use to describe graphs, but most are correlated and the relations between them are poorly understood and documented.

However, that is not a problem, we can figure out which structural properties matter by looking at all the primitive operations on graphs and seeing how the performance of their parallelisations correlates with properties.

Unless, of course, we don't really have a definition of "the primitive graph operations" either. It turns out that there is no consensus on what the core operations in graph processing are either. Thus, I found myself in a position with entirely too many degrees of freedom, leading top a floundering start.

> Every story has a beginning, a middle, and an end; but not necessarily in that order.
>
> Paraphrasing Jean-Luc Godard

At the heart of every thesis we find the divergence between what was planned and what was done. The difference between what we set out to discover and what was found. These two, often disparate, things need to be brought together.

After my experiences in ADVANCE I was fairly cynical about the quality and usefulness of the average computer science paper. I swore to abandon my stubborn idealism and focus on ruthless pragmatism. Do the bare minimum, lousiest possible engineering to get results and publish as soon as possible. Ignoring grandiose visions of polished tooling to automate benchmarking, analysing, and visualising performance data.

And, boy, was *that* a mistake. With all these degrees of freedom there was a lot of room to stumble into things that *don't* work. But negative results are hard to sell and publish. Especially when experiments are small-scale and ad hoc. I didn't start having a coherent story for this thesis until I started building my imagined tooling after all.

The final version of this thesis is completely dependent on this tooling. The "political/philosophical rant disguised as science" in Chapter 3, the performance analysis in Chapter 4, the machine learning in Chapter 7, and the portability study in Chapter 8 are all unimaginable without it.

Looking back at my version control history, the very first line of code for my tooling wasn't written until June 2017, exactly 3.5 years into my 4 year Ph.D. Imagine how much more useful and productive my time could have been, had I not decided to be "ruthlessly pragmatic" in 2014…

So, thank you, Ana. For your optimism and belief that all this floundering and effort would turn a coherent thesis, long before I did. Who knows where this work would have ended without it. And thanks Alex, for occasionally playing the "bad cop" to Ana's "good cop" about my work. I don't think I really understood some of the lessons you were trying to teach me early on, until I was solidly half way into writing this thesis.

And, of course, thanks to my committee for finding the time to read through this thesis and grill me at my defense: John Owens, Sven Scholz, George Fletcher, Paul Groth, Andy Pimentel, and Rob van Nieuwpoort. And thanks to Cees, who managed to put up with me for so long, and is undoubtedly happy to finally be rid of me!

Writing a thesis is stressful business in the best of times. Even more so, when done in addition to a fulltime job and a pandemic. So special thanks to Hare Koninklijke Hoogheid, Prinses Donder. Because nothing relieves writing stress like a purring cat or a friendly "*Prrrt?*".

And last, but certainly not least, Annelies. For putting up with the stress, frustration, and generally having to — vicariously — go through a second promotion. Sorry, schat; maar eindelijk klaar!

> I may not have gone where I intended to go, but I think I have ended up where I needed to be.
>
> Douglas Adams,
> *The Long Dark Tea-Time of the Soul*

CHAPTER 1

Introduction

> Everything starts somewhere, although many physicists disagree.
>
> Terry Pratchett,
> *Hogfather*

Can we improve the performance of graph processing using Graphical Processing Units (GPUs)? This seemingly straightforward question, is the starting point for the work in this thesis. However, before we consider this question, let us first consider the assumptions built into it. Why do we care about graphs and processing them? Why do we care about GPUs? Why do we want to combine these?

Graphs are flexible abstractions for describing relationships between discrete objects, making them well suited to modelling complex structures and/or relationships. As a result, graphs are widely used across different fields, such as linguistics, physics, chemistry, biology, bioinformatics, and social science.

Their usefulness puts graphs at the core of many computational problems in science and business. As we digitise more processes and data, the number and size of graphs people want to process will keep growing. This illustrated by the growth of and activity in the Graph500[1] ranking. The growing demand for graph processing is likely to continue for the foreseeable future.

Graph processing is not unique in this respect; demand for compute power has been growing in every field. This growing demand for processing

[1] http://www.graph500.org

power brings us to GPUs, which over the past 15 years have offered the most compute per dollar.

The design of modern commodity GPUs — with their massive, fine-grained parallelism and high-bandwidth memory — makes them very suitable as accelerators. Modern GPUs provide better computational throughput per dollar and/or watt than most traditional Central Processing Units (CPUs) and General Purpose Multi-Cores (GPMCs) architectures. They are the most cost-effective solution to increased compute demands, both in purchase price and FLOPS per watt. This is clearly shown by the TOP500[2] ranking, where the majority of supercomputers feature GPU accelerators.

In short, graph processing is important because many practical computational problems rely on it; GPUs are important because they are the most cost-effective way of meeting increasing demand for compute power, especially for massively parallel compute tasks.

In theory, graph processing should benefit a lot from GPU acceleration, as graph processing algorithms are characterised by large numbers of independent operations that can be parallelised, and high memory intensity [61]. However, as observed by Benjamin Brewster[3]: "In theory there is no difference between theory and practice, while in practice there is." [88]

1.1 Irregularity & GPU Acceleration

What makes it hard for graph processing to benefit from GPU acceleration *in practice* is, in short, our lack of understanding of the (parallel) performance of graph processing algorithms. This problem is compounded by the requirements imposed by modern GPU architectures.

There are some key trade-offs in the design of current GPUs that keep them affordable, while achieving the aforementioned fine-grained parallelism and high memory bandwidth — which we will cover in more detail in Section 2.2. The result of these trade-offs is that irregularity of memory accesses, branching behaviour, and poor locality drastically reduce the performance of code running on a GPU.

For programs with static access patterns, we can design code to play to the hardware's strengths to avoid these pitfalls. However, graph processing algorithms are highly irregular [61]. The data dependence of algorithms leads to unpredictable branching and poor locality. And the low computational intensity increases the impact of the (relatively high) latency of GPU memory.

These problems are not unique to either General Processing on GPU (GPGPU) or graph processing. The difficulty of modelling the performance of irregular algorithms is well-known within the High-Performance

[2] https://www.top500.org

[3] Not Einstein or Feynman, as the quote is often erroneously attributed. [72]

Computing (HPC) community. HPC practitioners generally have a lot of experience and intuition for mitigating these issues in their code — learned through years of practice.

Indeed, when looking at the state-of-the-art in GPU graph processing frameworks [17, 37, 45, 49, 66, 69, 97, 102] we see that each has its own approach to mitigate these pitfalls and exploit GPU features.

The problem with intuition, heuristics, and experience as tools is that they are hard to communicate, analyse, and validate. As a result, it is hard to quantify their effectiveness. This makes it difficult for experienced programmers and researchers to distinguish which of their intuitions and techniques are useful generalisations from experience, and which are superstitions or techniques that merely appeared to work by coincidence. It also means that it is hard to validate whether knowledge is still accurate for newer hardware platforms or whether it has become obsolete as the underlying hardware evolves.

It is not surprising that most papers address the impact of irregularity and mitigations of it empirically, rather than analytically. In graph processing we are dealing with four strongly intertwined dimensions: algorithm, data structures, hardware platform, and input graph. The impact of each dimension is dependent on the choices made for all the others. This makes isolating and testing the effect of choices along a single dimension difficult; in turn, this makes it hard to create analytical models that predict the optimal processing approach for a given problem.

In principle, there is no objection to an empirical approach to this problem. However, as we argue in Section 3.1 on page 18, the quality of empirical computer science often is not sufficient to support the claims made in papers.

1.2 Research Hypotheses

In summary, we care about graph processing because of its many practical applications and we care about GPU processing as it is currently one of the most cost-effective methods to obtain compute power. Additionally, several aspects of graph processing lend themselves well to the GPU programming model.

At the same time, the performance of irregular algorithms, including graph algorithms, on GPUs is poorly understood. There are no practical analytical models for GPU graph processing performance and the available empirical results are based on datasets that are too small [3, 46, 61]. As a result, a significant part of this thesis deals with developing our own analytical models and empirical methods for tackling the problems of GPU based graph processing.

Going into this research we had a number of vague intuitions about the behaviour of graph algorithms and GPU behaviour, which are best captured by the following two hypotheses:

1. Graph structure has a significant impact on GPU graph processing performance, and
2. this impact is relatively stable across GPU generations.

The goal of this thesis is to develop analytical and empirical methodology for investigating the link between graph structure and parallelisation strategy for graph processing on the GPU. We demonstrate the effectiveness of our methodology using the PageRank and Breadth-First Search (BFS) algorithms as case studies. Both PageRank and BFS are commonly used to benchmark graph processing systems, allowing for comparisons with existing solutions.

To validate our hypotheses, we identify the following five objectives:

1. Build tooling to produce, aggregate, and analyse the performance data of multiple algorithms, different parallelisations of each algorithm, across multiple hardware platforms, and across different input graphs.
2. Quantify the performance impact of graph structure on the performance of the various parallelisation strategies and their data representations, and show that there is a significant performance impact.
3. Model the relation between graph structure and parallelisation strategy, allowing prediction of the best parallelisation strategy for a given graph.
4. Show that it is feasible to exploit the relation between graph structure and parallelisation strategy to improve on the performance of the state-of-the-art.
5. Show that the relation between graph structure and parallelisation is, indeed, stable and can be exploited across GPU generations.

1.3 Scope of Thesis

We are faced with a performance engineering challenge. To improve the performance of GPU graph processing, we need to untangle the complex interactions between algorithms, data representation, hardware, and input graphs. Unfortunately, as alluded in the previous section, there are no adequate analytical models, and there are insufficient empirical results to start from.

Our goal for this thesis is to establish a foundation for both analytical and empirical investigation of the interaction between algorithm, data representation, hardware, and input graphs. However, addressing this interaction in its full generality is far too broad a scope for a single thesis. Even examining each dimension in isolation we find a lot of unknowns. In this section, we set out how and why we limit the scope of our investigation for each of these dimensions.

1.3.1 Algorithms

HPC graph processing papers often talk about the ubiquity, flexibility, and importance of graphs and graph processing algorithms. Given their ubiquity, it is surprising that there is no consensus on which real world applications to use as benchmarks for graph processing systems.

Most of the literature uses well-known, but simple algorithms such as BFS, Single-Source Shortest Path (SSSP), PageRank [74], Betweenness Centrality (BC), and various clustering algorithms. The Graph 500 [24] benchmark, for example, only concerns itself with BFS and SSSP. Yet it is unclear to what extent these algorithms are representative for real world algorithms used in model checking, bioinformatics, or network analysis.

This lack of consensus extends to the idea of "primitive" graph operations — i.e., set of operations on which all graph algorithms can be based. Neighbour iteration and common neighbour iteration are both operations that are simple and frequent enough to qualify, but for other operations it is less clear. For example, in model checking, BFS and Depth-First Search (DFS) are often considered primitive operations used as a part of other algorithms, whereas most HPC papers treat them as complete algorithms.

Establishing a minimal set of primitive operations makes it feasible to comprehensively evaluate implementation strategies for each. Additionally, it would limit the scope along the "algorithm" dimension, making it simpler to analyse how it is affected by the other dimensions of our problem.

In this thesis we opt to focus on the use of neighbour iteration as a primitive operation. It is simple to implement, simple to formalise, and widely applicable in the implementation of graph algorithms. As such, results related to neighbour iteration are applicable to many graph processing algorithms. Specifically, we examine neighbour iteration in the context of BFS and PageRank[74]; both algorithms are commonly used in the literature.

1.3.2 Data Representation

Data representation — i.e., data structures and in-memory layout — and parallelisation strategy are closely related. The number of parallelisation strategies that make sense for a specific data representation (or vice versa)

are often limited. On GPGPU platforms, the hardware further restricts which parallelisation strategies, and thus data structures, are sensible to use.

As a result, this is one axis where the scope of our investigation is automatically limited. In Section 2.3 on page 14 and Section 4.2 on page 43 we discuss possible parallelisation strategies for graph processing on the GPU. We conclude that there are only a limited number of sensible parallelisation strategies, limiting our data structure choices to those parallelisation strategies. For the parallelisation strategies we use in this thesis, the relevant data structures are: edge list, Compressed Sparse Row (CSR), and several minor variations of these.

1.3.3 Hardware Platform

The core programming model of GPUs has been the same for the past 15 years. However, there have been many changes and advances in the internal logic of GPUs, which affect thread scheduling, access coalescing, atomic operations, caching behaviour.

It is unclear how big the impact of these implementation details is on the performance of graph processing code. As a result, we don't know how much our optimisation techniques, intuition, and code are portable across hardware platforms/generations, both past and future.

In this thesis we use four different types of NVIDIA GPUs for comparisons across hardware generations. We use Kepler architecture K20 cards from 2012, Maxwell architecture GTX980 cards from 2014, Maxwell architecture TitanX cards from 2014, and Turing architecture RTX2080 Ti cards from 2018.

1.3.4 Input Graphs

We know that the structure of input graphs affects the performance, but little is known about which structural properties are relevant or how big their impact is. Some of the more important properties are: edge count, vertex count, degree distribution, diameter, clustering coefficient. There is a near endless variety of other structural properties, but there is no consensus on which properties best characterise the "structure" of a graph.

One of the reasons for this lack of consensus is that most of these properties are strongly correlated. This makes it hard to classify graphs by "structure" and investigate how that impacts performance. For example, how do we determine if a given graph dataset is biased to graphs of certain types of structure, if we do not know how to identify those structures?

In this thesis, we explore two different approaches to tackle the selection of input graphs. First, we explore the generation of graph datasets that contain graphs with sufficiently different structure (see Chapter 5 on

page 67). However, the strong correlation of structural properties makes this a challenge. Second, we used graphs from publicly available graph repositories. As such, the scope of our thesis is limited by the success of our graph generation and the (public) availability of graph datasets.

1.4 Thesis Structure & Contributions

Chapter 2 on page 9 presents an overview of graphs, GPGPU programming, and graph processing on GPUs. Providing the necessary background information for the discussion in the rest of this thesis. Most importantly, we include a more detailed overview of the design trade-offs present in modern GPUs and how these constrain our implementation choices for graph processing.

Chapter 3 on page 17 discusses the reproducibility difficulties of empirical computer science, with a focus on the difficulties in graph processing. We present the software toolchain, published in [89], we built for gathering performance data of graph algorithms, running data analyses on these results, and evaluating models against our empirical data.

In Chapter 4 on page 41 we discuss possible parallelisation strategies for neighbour iteration and how neighbour iteration can be used to implement the PageRank and BFS algorithms. We use our toolchain from Chapter 3 on page 17 to quantify the performance impact of different parallelisation strategies on the performance of PageRank, published in [92, 93], and BFS, published in [94].

Chapter 5 on page 67 presents our graph generator based on evolutionary computing, published in [95]. We built this generator as a proof-of-concept to generate input graphs for our experiments. However, we eventually abandoned this approach in favour of using real-world datasets, due to pragmatic constraint on engineering time.

Chapter 6 on page 85 presents analytical workload models, published in [94], for the parallelisation strategies from Chapter 4 on page 41. We demonstrate that these analytical models are not sufficiently accurate to predict parallel performance of these algorithms on the GPU.

In Chapter 7 on page 99 we show how we can use our toolchain from Chapter 3 on page 17 to create Binary Decision Tree (BDT) models that let us predict the best implementation of an algorithm for a given input graph, published in [92, 93]. Furthermore, we demonstrate that we can speed up BFS traversals by using our BDT models to dynamically switch implementations during a traversal.

Chapter 8 on page 111 presents an analysis of the portability of our BDT models across datasets and GPU architectures, published in [90]. Our analysis shows that the performance characteristics of our implementations are largely stable across GPU architectures and datasets.

CHAPTER 2

Background

In this chapter, we provide a brief introduction to graphs, General Processing on GPU (GPGPU) programming, and graph processing on Graphical Processing Units (GPUs). The main goal is to introduce the concepts, terminology, and notation necessary to understand the remainder of this thesis.

2.1 Graphs

Graphs are an abstraction for describing relationships between discrete objects. Thus, graphs are a natural fit for working with many of the complex, relational datasets we encounter in the real-world. As a result, graphs are widely used across different fields, such as (computational) linguistics, physics, chemistry, biology/bioinformatics, and social science.

A graph describes a collection of entities (called *nodes* or *vertices*) and the relationships between these entities (called *edges*). We can, optionally, ascribe one or more attributes to these relationships. A real-world example of a graph would be that of a rail network: each vertex represents a station, each edge represents a rail connection between two stations; distance, travel time, and/or capacity represent examples of edge attributes.

Formally, $G(V, E)$ denotes a graph G, a set of vertices V, and a set of edges E. An edge $(u, v) \in E$ represents a binary relationship between vertices $u \in V$ and $v \in V$. We can distinguish several subcategories of graphs within this definition — such as directed graphs, undirected graphs, and multigraphs.

Traditionally, the assumption is that graphs are undirected. Edges are unordered pairs, meaning that $(v, u) \in E$ and $(u, v) \in E$ are equivalent.

Both mean that there is a connection from u to v and from v to u. Directed graphs are considered a related, but different abstraction. For directed graphs we consider edges to be ordered pairs. Therefore, (v, u) and (u, v) are distinct. Thus, $(v, u) \in E$ means there is a connection from v to u, but not vice versa.

In this thesis we use a different viewpoint. We assume graphs are directed — i.e., edges are ordered pairs. From this viewpoint, undirected graphs are isomorphic to the subset of directed graphs where:

$$\forall v, u \in V \,.\, (u, v) \in E \iff (v, u) \in E$$

Mathematically, these two viewpoints are equivalent, but, from an implementation perspective, the latter is more convenient: when we treat undirected graphs as a subset of directed graphs, we only need to write an implementation for directed graphs without needing to introduce special cases and edge conditions for undirected graphs.

Multigraphs are another common subcategory of graphs. A multigraph is a graph where the edge (u, v) can appear multiple times, i.e., there are multiple connections between vertices u and v. This can be used to, for example, model multiple independent interactions between two entities.

We do not consider multigraphs in this thesis, as, in most cases, we can reduce multiple edges between vertices to a single edge with an attribute storing the number of duplicate edges. Furthermore, for structural algorithms, like traversals, having multiple connections between two vertices results in behaviour identical to the case of a single connecting edge. There are some algorithms and uses of multiple edges that cannot be replaced this way, but we do not concern ourselves with these in this thesis.

2.2 Graphical Processing Unit

The purpose of a GPU, as the name suggests, is processing graphics. The transition to real-time 3D graphics in video games in the early to mid nineties led to consumer demand for hardware 3D graphics acceleration.

Initially, GPUs were sold as separate accelerator cards, which is what most people think of when talking about GPUs. Nowadays, even devices without a separate graphics card have some form of hardware acceleration for 3D graphics, usually integrated as a part of the device's System-on-Chip (SoC)[1].

Accelerating 3D graphics and rendering pixels to a screen involves a lot of embarrassingly parallel tasks. People quickly realised that many High-Performance Computing (HPC) and scientific computing tasks could benefit from exploiting this parallelism. For example, in 2001 Larsen and McAllister used a GPU to implement fast matrix multiplication [53].

[1] For example, in modern smartphones.

At first, this use of GPUs relied on tricks to encode the computation as part of the graphics pipeline. For example, in [53] the input matrices were encoded as texture images, with the result matrix encoded in the rendered pixels. As more people started using GPUs for non-graphics computation, there was an incentive for manufacturers to accommodate this use.

In November 2006, NVIDIA announced the Compute Unified Device Architecture (CUDA) programming model. CUDA allows programmers to write code that can be run directly on the GPU, rather than having to resort to hacks and workarounds. In December 2008, the Khronos Group released the Open Computing Language (OpenCL) specification, the result of a collaboration between Apple, AMD, IBM, Intel, NVIDIA, and Qualcomm. The goal of OpenCL is to provide a cross-platform model for programming GPUs and other massively parallel accelerators.

Most of this thesis' implementation work was done using NVIDIA's CUDA platform. Our choice to use NVIDIA's proprietary standard over the open OpenCL standard was a matter of pragmatism. At the time, there were no publicly available implementations of OpenCL 2.0, and the CUDA developer tools were[2] easier to use and higher quality.

It should be fairly straightforward to translate both our code and research methods to OpenCL, as the core programming concepts of CUDA and OpenCL are largely the same. As such, this choice mainly affects the terminology we use. We opt to use CUDA's terminology, as it matches our code and is already widely used in GPGPU programming communities.

2.2.1 The GPU Design

There are some key trade-offs in the design of GPUs. The overarching theme of these is: *Throughput over latency.*

The most obvious of these is the cores on the GPU. There is a trade-off between the raw power of individual cores and the number of cores that can be packed on the chip. The GPU cores, called Stream Processors (SPs), are slow compared to modern desktop Central Processing Units (CPUs). Clock rates range from 700 MHz to 1.7 GHz, in contrast to the 2.4 GHz and up seen in modern desktop/server CPUs.

Slower and simpler cores enable GPU manufacturers to put more of them on a single GPU. It is not uncommon for modern GPUs to have hundreds or thousands of parallel cores. This is a performance win *iff* there is enough parallel compute work to keep all cores busy.

The other key trade-off concerns memory. GPUs have memory with high bandwidth, but also a high latency for memory accesses. This high latency is hidden by using Simultaneous Multi-Threading (SMT) [96]. With

[2] And largely still are…

SMT we keep multiple instruction streams per core; when an instruction stream is stalled waiting for memory, the core simply switches to another instruction stream. As long as there are enough parallel instruction streams, there will always be useful work to do while data is fetched from memory. The high bandwidth is exploited by coalescing multiple smaller memory fetches into a single bigger memory fetch.

GPGPUs are programmed using a Single Instruction, Multiple Threads (SIMT) programming model. The individual SPs are grouped into Streaming Multiprocessors (SMs). The SM schedules and executes groups of threads called *warps*. There are always 32 threads within a warp[3], but the number of SPs per SM varies over hardware generations.

The cores within an SM share a single instruction scheduler, which executes all threads within a warp in lockstep. To accommodate SIMT latency hiding, each SM stores the registers and entire execution context of multiple active warps. As a result, there is no overhead to context switching between active warps [70, Section 4.2]. When one warp is stalled on a memory fetch, another warp can be computing.

A result of this SIMT model and shared instruction scheduler is that all threads in a warp execute the same instructions. This would seem to rule out the use of dynamic branching instructions within a kernel, since threads within a warp might end up executing diverging branches. Instead, diverging branches are handled via the SM's instruction scheduler. This scheduler can disable specific threads in the warp. Divergent branches are handled by executing both of a branch's code paths in order, disabling all threads in the warp that took the other branch. The downside of this solution is that the efficiency of code with divergent branches is reduced, as one or more of the cores of the SM are not doing any computation during half of that branch.

NVIDIA's latest Volta architecture [71] reduces the impact of branch divergence. It allows arbitrary interleaving of instructions from each code path, rather than executing the code paths one after the other. This allows SMT-style latency hiding between threads within the same warp, rather than just between entire warps and thus reduces the efficiency loss from divergence.

While lockstep execution reduces efficiency of divergent branches, it allows the hardware to coalesce adjacent memory accesses by threads within a warp into a smaller number of large fetches. Coalescing reduces the number of memory fetches required, and, thus, the time that threads are stalled waiting for these fetches to complete.

[3] At least, in all NVIDIA GPUs so far.

2.2.2 The GPU Memory Hierarchy

The GPU's Non-Uniform Memory Access (NUMA) architecture is considerably more complex than that of the host. On the CPU there is only global memory and three layers of caches, with the caches preserving consistency. In contrast, GPUs have a weak memory consistency model and multiple different types of memory. The four relevant types of memory are: global memory (and caches), constant memory, texture memory, and per-block shared memory.

Global memory is the slowest of these memory types. Global memory can be read or written by any thread on the device. However, the weak memory consistency model means that there are no guarantees about the observed order of reads and writes from different threads. Programmers have to use explicit barriers and memory fences to ensure that reads and writes from other threads or other blocks are visible within the current thread. There are L1 and L2 caches available to reduce the latency of global memory, but their usage often requires extra configuration.

Constant memory refers to a logically distinct part of global memory. This part of the address space has its own dedicated, read-only cache and logic to reduce the memory traffic required when the data is read by many threads simultaneously. This makes constant memory suitable for run time defined, read-only constants shared across every thread in the kernel invocation.

Texture memory, like constant memory, is a logically distinct portion of global memory. There is a dedicated cache for texture memory which, unlike constant memory and its cache, is not limited to a dedicated part of the address space. Instead, programmers use specific texture functions to perform reads from texture memory. The cache is optimised for 2D spatial locality, rather than regular memory locality. Additionally, addressing calculation and optional value interpolation happen in dedicated logic units, reducing compute work for the SPs.

Per-block shared memory is directly attached to the SMs, which means it is only accessible from a single SM, but provides much higher bandwidth and lower latency than global memory. It is made up of multiple parallel memory banks, so accesses spread over multiple banks can be handled in parallel, increasing the bandwidth further. The lifetime of values in shared memory is limited to the lifetime of the active block running on the SM, unlike global, constant, and texture memory whose values persist across kernel invocations.

2.2.3 Programming GPUs

Programs using GPGPU processing consist of two parts. Part one is a normal program running on the CPU, also called the *host*. The second

part is the code running on the GPU, also called the *device*. Individual parts of the device code are called *compute kernels*, or just *kernels*.

Kernels are usually written in an extended version of C, such as CUDA C/C++ for NVIDIA or OpenCL C for the various OpenCL implementations. But we are slowly seeing more high-level languages and Domain Specific Languages (DSLs) for GPGPU computing [21, 39, 42, 76, 85]. These languages generate either their own CUDA/OpenCL code, or assembly for the Parallel Thread Execution (PTX) Instruction Set Architecture (ISA) exposed by NVIDIA.

The main difference between a GPGPU kernel and a regular function, besides running on a GPU, is that for the kernel we execute many independent instances of the same code, rather than just one. To do so, when launching a kernel from the host code, we must specify a 3-dimensional *grid* of threads to run.

Each thread is assigned a tuple of x, y, and z indices within the grid. These indices are accessible within each thread and are used to build a mapping between threads and data, such that different threads execute the same instructions on different data.

The programmer also specifies how to further split this 3-dimensional grid into 3-dimensional *thread blocks* of equal size. These blocks are used to distribute threads over the available SMs. Within a block, the hardware further splits the threads into warps. An SM holds all warps within a block simultaneously, allowing zero cost context switching between them.

2.3 Graph Processing on GPU

The flexibility of graphs as a data structure means there are endlessly many ways to implement graphs algorithms on a CPU. But on the GPU, the limitations imposed by the hardware restrict the number of viable implementations for any given algorithm.

In this thesis we restrict ourselves to a Bulk Synchronous Parallel (BSP) execution model. In a BSP model, an algorithm consists of a number of supersteps with barrier-synchronisations in between. BSP is a natural fit for the GPGPU kernel interface, and most graph algorithms can be restated in terms of iterative steps.

These steps generally boil down to performing some operation on each element in a set of vertices or edges. This set can be the entire graph, a subset of the graph, or the *frontier* of a traversal.

Graph processing is characterised by a low arithmetic intensity and large number of independent operations [61]. The large number of independent operations means there is enough work to parallelise across the many cores of a modern GPU. However, the low arithmetic intensity in

graph processing means there is very little compute work for these cores to do.

Thus, the key to efficient graph processing on GPU is to focus on maximising the use of memory bandwidth. In the previous section we discussed the SMT-style latency hiding used between warps on a single SM. In the ideal scenario the entire time between issuing a memory fetch and the fetch completing can be filled with issuing fetches for the other warps. This is something that should be kept in mind when considering parallelisation and workload distribution.

The need to saturate each SM with enough parallel work to effectively perform latency hiding restricts the number of sensible parallelisation strategies. We can do edge-centric parallelisation, using one thread per edge, or vertex-centric parallelisation, using one thread per vertex. In the latter case there are two possible variations: we either *push* updates for each outgoing edge of a vertex, or we *pull* updates from each incoming edge of a vertex.

When dealing with a vertex subset or frontier, there are two further options. One option is to use a Gather-Apply-Scatter (GAS) based approach. This involves gathering the relevant edges or vertices into a new data structure, applying the above parallelisation strategies to process this data, and, finally scattering the results back to the original data structure. The alternative option is to always launch threads for every vertex or edge in the graph, and set the threads for vertices/edges outside the active subset to run no-ops. This has the benefit of skipping the copying of the gather and scatter phase, but reduces the efficiency of the SMs by having no-ops threads scattered across warps.

Vertex-based parallelisation strategies often perform fewer redundant reads and/or fewer atomic operations than edge-based ones (see Chapter 6 on page 85). However, they are susceptible to efficiency loss due to load imbalance. Suppose we have a vertex with one million edges that is processed by a thread in a warp where every other vertex has five edges. In this scenario, 31 threads in the warp will be idle until all million edges have been processed in the remaining one thread.

Ultimately, how successfully a GPU graph processing algorithm saturates the bandwidth depends on the coalescing of memory accesses. The result is that there is no single best parallelisation strategy for any algorithm or graph, as the chosen strategy only partially controls the memory locality and access pattern. The rest of the locality and access pattern is dependent on the graph's structure and specific algorithm.

CHAPTER 3

The Art & Engineering of Empirical (Computer) Science

Ostensibly, this thesis is about General Processing on GPU (GPGPU) programming. However, in reality only 770 (2.96%) of the 26,036 lines of code covered by this thesis are Compute Unified Device Architecture (CUDA) code. Even if we broaden our definition of GPGPU code to include all the host code and the build system, it is still only 7,650 (29.38%) of the total code.

The remaining 70.62% (or 97.04%) of the code exists solely to deal with running experiments, tracking provenance of results, postprocessing data, validating results, and performing data analyses. In other words, it is "just" engineering work that is normally seen as incidental to the pursuit of "actual" science. The purpose of this chapter is twofold:

First, to explain the design of and rationale behind the data format we use for our results, give an overview of the tools we developed for running experiments, tracking the provenance of our results, and performing analyses.

This chapter is based on work previously presented in:

Merijn Verstraaten. *Belewitte*. Version 1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6959684. URL: https://doi.org/10.5281/zenodo.6959684

Merijn Verstraaten. *Belewitte GPU Experiment Results*. Version v1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6925023. URL: https://doi.org/10.5281/zenodo.6925023

Second, and more importantly, to motivate why our data format and tooling are *not* just engineering. We argue that:

- The data format and tooling are an essential contribution of this thesis,
- the design and implementation of data formats and tooling are inherently part of the scientific process, not "just" engineering, and
- this engineering work should be an expected part of any performance engineering or other experimental computer science.

3.1 Motivation

Performance engineering is the art of untangling the complex interactions between hardware, data structure, and algorithms to achieve the best possible performance. The challenge of performance engineering for GPGPU graph processing is at the core of this thesis.

Each of the areas of hardware, data structures, and algorithms covers decades of development and layers upon layers of abstraction, on both the hardware and software level. And all these areas and abstractions influence each other in indirect ways. As a result, a complete and accurate simulation of the interaction(s) between all relevant components is infeasible.

In other words, in performance engineering we cannot feasibly work from "ground truth". We can only observe how systems behave in the real world, model this behaviour, and verify that our model approximates reality sufficiently to be useful.

This can feel limiting in a field where theoretical research has the luxury of working with well-defined mathematical abstractions, but it is no different from the daily reality in other empirical fields like experimental physics, astronomy, or chemistry.

If we, computer scientists, want to take ourselves seriously as a science, we should aspire for our research to be at least as principled as these traditional empirical fields.

3.1.1 The State of Empirical Computer Science

In 1980 Denning, then president of the ACM, already argued that "experimental" computer science is an empirical science and that we should evaluate according to the same standards as more traditional empirical sciences [27].

25 years later he admonished us — computer scientists — for not heeding his warning [28]. In this article he indirectly references[1] two studies

[1] via the 1998 article [87] by Tichy

on experimental validation of pre-1995 and pre-1998 computer science papers [86, 100]. These studies conclude that, after excluding theoretical and mathematical papers, 30–40% of computer science papers fail to have any experimental validation.

The situation seems to have improved since then, but there are still plenty of problems. The real litmus test for (empirical) science is whether others can reproduce the results. And just because a paper describes an experimental validation, does not mean it can be reproduced. A 2015 study on reproducibility by researchers at the University of Arizona found that they could only reproduce 54% the experiments of 402 papers involving experimental validation [23].

These findings mimic our own experience: while some authors take care to make their code publicly available and archived on Research Data Management (RDM) platforms — such as Zenodo [18] — other research software simply gets lost. At least one author we approached indicated that the graph-processing code we were interested in using was lost in the process of moving between different institutions.

Problems with reproducibility are not the only issue in empirical computer science. The quality of the produced empirical data varies greatly from paper to paper. A recurring problem we have observed in papers on GPGPU graph processing is that the conclusions are justified based on very small result datasets — i.e., results from 5–20 graphs — in most cases.

Extrapolating from such small samples is fine if the problem is regular or when samples are truly representative of the wider data. However, GPGPU graph processing is not regular and these sample graphs are not representative of all graphs — or, at least, we have no way of knowing if they are.

We do not blame the authors for this, there is only so much room for graphs and tables included directly in a single paper. Summarising large benchmark datasets in a way that is both complete and understandable for the reader is a challenge, and computer science does not have a tradition of separately publishing datasets of empirical results.

Furthermore, there are no standard classification schemes or metrics for evaluating the similarity of (sets of) graphs. This makes it hard to assess how similar two graphs are or how representative a set of graphs is off all possible graphs. This is a problem that crops up in Chapters 4, 5, and 8 on page 41, on page 67, and on page 111, too.

3.1.2 The Wider Empirical World

Fortunately, we are not alone in this. In a 2016 survey by Nature, over 70% of the 1,576 surveyed scientists said to have tried and failed to reproduce work by others. When asked which factors contribute to these failures, over

40% of the respondents indicated that unavailable code, methods, and raw data are "often or always" part of the problem.

The underlying causes of reproducibility issues impact science in other ways too. 2016 also saw the publication of the FAIR guiding principles for scientific data management [98]. The paper argues that we can extract more scientific value from scientific data if we improve the ecosystem(s) for publishing scientific data. The authors highlight the value of archiving, sharing, and linking datasets in ways that allow other researchers to find, (re)use, and build upon them. They further discuss the problem areas that currently make this difficult. The key problem areas they identified have significant overlap with the problems identified by the Nature survey respondents [6].

The key take away of the paper is that datasets produced by empirical experiments have value beyond the papers that gave rise to them, but only when others can find, use, and refer to them. Which is what they hoped to capture with their FAIR acronym: Findability, Accessibility, Interoperability, and Reusability.

The paper started discussions in many empirical fields and has already had an impact, as many funding agencies — including NWO[2] — now require data management plans and proper data archiving for all projects they fund.

Of course, the concerns about FAIR data are equally relevant for research software[3]. This led to a reformulation of the FAIR principles tailored to software [52].

3.1.3 Practice What You Preach

Fortunately, the situation is improving. Computer science conferences are increasingly employing "artefact evaluation" committees to evaluate software artefacts, source code is increasingly publicly available, work on making software citable [29, 79], and funding agencies are starting to require proper archiving of both data and software.

However, there is still plenty of room for improvement: code is often available on faculty websites or GitHub, rather than scientific archiving organisations, such as Zenodo [18]. Papers often do not clearly indicate the exact software versions used for experiments, and it is still very rare for computer scientists to publish datasets of empirical results.

The above factors make it difficult to explicitly cite or credit software and datasets produced by other researchers, despite the considerable effort and engineering that has gone into them.

[2] https://www.nwo.nl/en/research-data-management

[3] Research software is defined here to mean "software whose primary purpose is the generation or validation of data for use in scientific research."

Following our own arguments above, the exact code [89] and data [90] used in thesis are available under the General Public License version 3 (GPLv3) [34] and Creative Commons Attribution 4.0 International [26] licences respectively. They are archived on Zenodo, with the latest version being available at `https://github.com/merijn/Belewitte`.

3.2 Data Format Design & Refinement

In the previous section we argued the importance of publishing datasets of results in addition to publishing the software used to generate those results. But, if that "dataset" is merely a ZIP file of unlabelled CSV files, this is not useful to anyone. This is reflected in the FAIR acronym.

The first half, findability and accessibility, deals with unique identification of datasets, metadata availability, and getting access to a copy of the dataset. These concerns can be addressed by using a dedicated RDM service for archiving and sharing datasets.

The second half of the acronym, interoperability and reusability, deals with more nebulous concepts. In [98], reusability is clarified to refer to: "having a clear licence for data and metadata usage, detailed provenance of the data and metadata" and "meeting domain-relevant community standards". Interoperability is clarified as: "references other data using qualified references" and using a "formal, accessible, shared, and broadly applicable language for knowledge representation".

While these statements give us no hard definition, the intent is clear:

- Use standard and commonly used data formats that others can access without too much effort,
- ensure the dataset or schema clearly defines the data stored, and
- include all the metadata necessary to track the provenance of results to their experiments.

In other words, the recommendation is: "use what everyone else in the field is using". Fields where the publication of datasets is already common — such as climate science, ocean science, or computational chemistry — have settled on a handful of widely available formats that everyone uses. However, publishing result datasets is rarely done in computer science, so there is no real established format yet, requiring us to invent our own format.

Designing a data format sounds simple enough: decide what data you need to store and write down a schema for it. Our experience has taught us that, in practice, it is not obvious which data we need. Thus, the design of a data format is an integral part of our research's refinement process.

As shown in Fig. 3.1, we start out with research questions in mind. Based on these questions and our assumptions, we can formulate what data we need. Once we know which data we need, we can run our experiments and gather the data. Analysis of the data rarely answers our research question directly. Instead, we find ways we need to refine our hypotheses and questions, which leads to re-evaluation of what data we need, etc. This loop continues until we are satisfied with the answers to our questions[4].

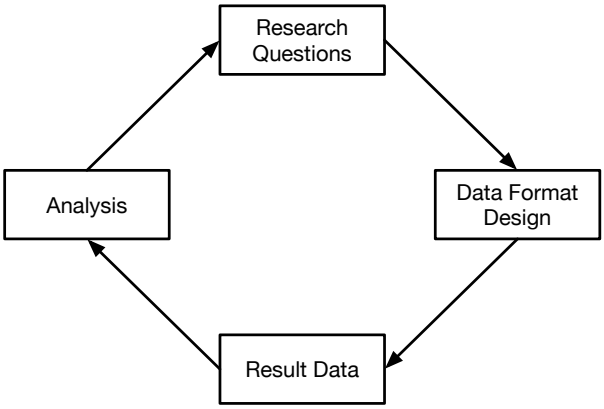


Figure 3.1: Data format design as integral part of scientific inquiry.

A pragmatist might argue that refining the data format is a redundant part of this refinement loop: "just store everything". Which is a simple enough idea, but even deciding what "everything" entails can be a challenge. To illustrate these difficulties, the rest of this section presents an overview of how our data format has evolved over time — i.e., the difficulties we encountered in designing a data format, how we addressed the oversights we discovered, and the importance of accommodating future refinements into it.

3.2.1 The Messy Initial State

During the initial stages of this project, like many scientific endeavours, the code consisted of a collection of Python and bash scripts to run experiments, collect, and analyse the data. Similarly, the "data format" was, essentially, piles of undocumented CSV files. This "works" if there is only one person manually investigating data from a handful of experiments, but when the number of experiments grows, this approach becomes untenable.

We started running into problems once we started to benchmark larger datasets, in an attempt to address our complaint that many GPGPU graph

[4] Or funding runs out, whichever comes first…

processing papers base their conclusions on rather small datasets. Many of our scripts relied on manual pre- and postprocessing of data. Runs on large datasets across several Graphical Processing Units (GPUs) led to an explosion in the number of result files, approaching 50,000 result files at one point.

Each change to the GPGPU code required redoing everything to obtain the new timings and results, making the process even more labour intensive and error prone. The combined challenge of massive results collections and frequent code changes made it extremely difficult to manage the experiments and track the provenance of our results.

3.2.2 An Initial Schema

Our goal was to kill two birds with one stone: reducing the manual effort and time required to do experiments and keep track of the results, while also making it easier for others to build on top of our code.

A large part of the manual intervention, pre-/postprocessing scripts, and time investment dealt with querying, aggregating, and filtering our data into subsets for analysis. In other words, the sorts of operations that SQL is well suited for. Additionally, the provenance information we need to store to link results to experimental configurations maps well to a relational model.

We decided to use SQLite [81] as basis for our data format. It gives us all the power and convenience of a relational database and SQL, but does not require us (or other researchers) to set up, configure, or maintain a separate database server to use it.

Since SQLite databases consist of a single file, it is easy to archive, backup, or share the entire dataset of results and their provenance. It is also a well-known, well-supported, and open file format that is supported by practically every programming language. There are standard tools available for operating on SQLite databases, and data can be easily exported to common formats such as CSV.

Finally, SQLite is one of the formats recommended by the US Library of Congress [25] for the long term archival of complex datasets.

The schema of the first version if our data format is shown in Fig. 3.2 on page 25 and covers:

Platform
: The platform timings were performed on.

Algorithms
: The set of algorithms we implemented and benchmarked.

Implementations
: The different implementations we have of each algorithm.

Graphs
The input graphs we run our benchmarks on.

Graph properties
Structural properties of our input graphs.

Variants
Some algorithms can be run in multiple configurations (such as root nodes for Breadth-First Search (BFS)) on the same input graph; a variant identifies a specific combination of graph, algorithm, and configuration.

Step properties
Runtime properties for specific supersteps of a variant (such as frontier size for BFS).

Total time
Global timing results for a single variant as measured using a single implementation on a single platform.

Step time
Timing results for individual supersteps of a single variant as measured using a single implementation on a single platform.

The main catalyst for our initial schema was our difficulty with tracking the provenance of results and the performance of our data processing. As a result, the first version of our data format served mainly to aggregate and store the results of manually performed experiments — and worked well for these problems, although it had quite a few oversights and omissions.

As an important side-effect, the SQLite database was also more efficient: it took up considerably less disk space than the thousands of text files required otherwise, and the analysis time went down from ∼40 minutes to ∼1 minute.

3.2.3 Data Format Refinement

Because the first version of our data format was a big improvement over the ad hoc scripts before it, we started using it more intensively for all the tooling related to our experiments, which we will discuss later on in Section 3.3 on page 30.

We spent considerable effort on the design of the initial schema, leaving room for several extensions that we wanted, but did not immediately need. Despite this effort, we discovered oversights and omissions as we started using our new tooling more.

Most of the omissions were related to metadata. For example, we stored which input graphs were used for experiments, but not which dataset each

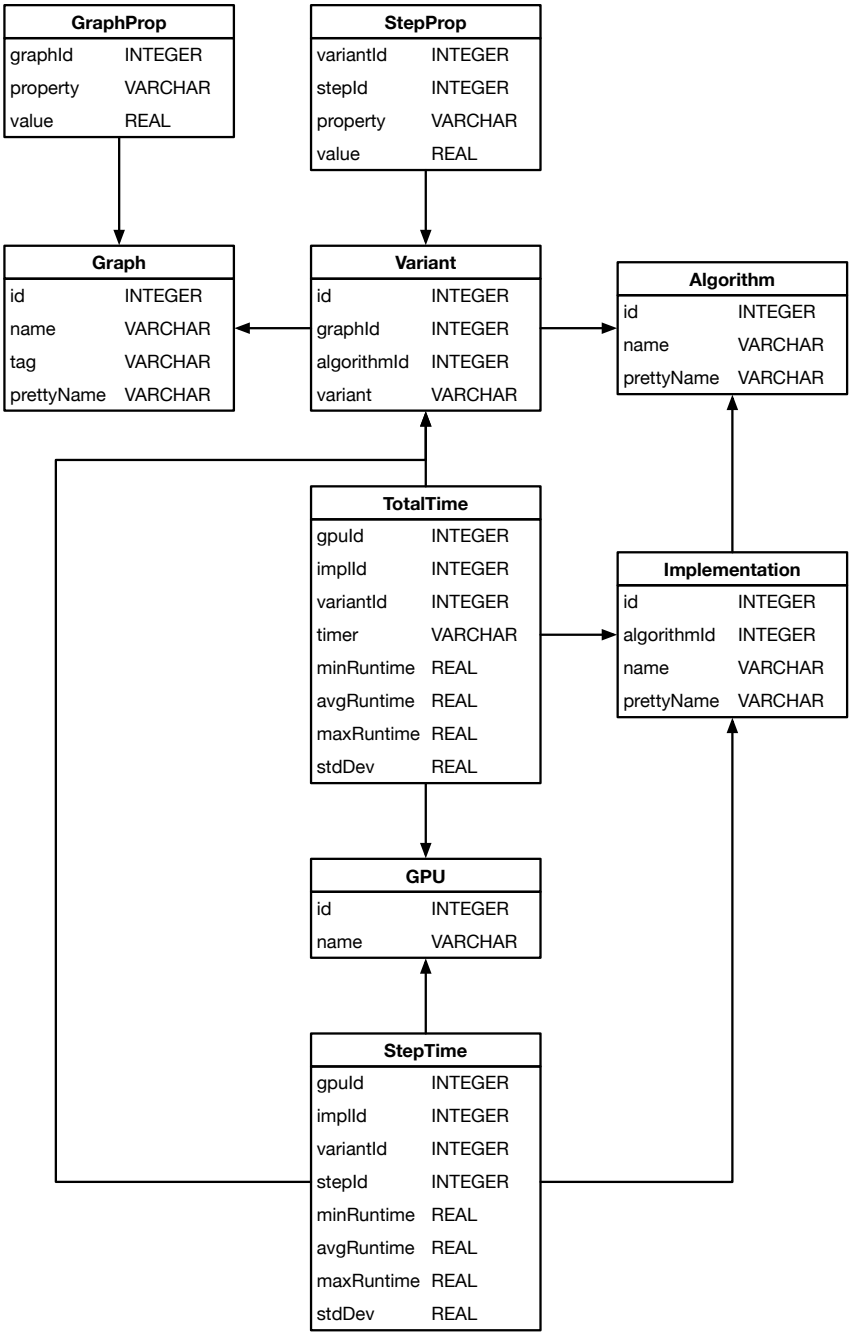


Figure 3.2: Schema for the initial version of our data format.

graph came from. This made it hard to compare our results across multiple datasets. Similarly, we did not store or verify which specific version of our GPU code was used for experiments, which led to one of our databases containing "contaminated" data — i.e., results from multiple different versions of our GPU code. Requiring us to rerun all those experiments to obtain the dataset used throughout this thesis. Nor did we store the number of repeated experiments our results were averaged over.

The biggest omission in the initial data format was that we did not store the results of our analyses and modelling code in the database. The resulting models were stored on disk, losing all training metadata. This training metadata cannot be reconstructed, as our model training algorithm is stochastic.

From the initial version onwards there has been a continuous loop of tooling to make experiments or analyses easier, leading to increased usage and reliance on these tools, running into oversights and omissions, fixing these problems, and then even more reliance on the tooling.

During one of the first refinements to our schema, we made the fortuitous decision to implement a versioning scheme for our schema, and implement migration logic from older schemas to the latest version. As a result of this effort, even databases created with the first iterations of our code can still be read and processed by the current version after an automated migration[5]. This has been invaluable in addressing omissions in the schema and extending the functionality of our tools.

In the ∼2.5 years since the initial version of our data format, the schema has been refined extensively (we are currently on the 29th version of the schema), and our tooling expanded. The result of this effort can be seen in the latest version of the schema, shown in Fig. 3.3 on page 29.

In summary, we rectified the metadata omissions discussed above, and added the ability to import and store results from external implementations for easier comparison with work by other researchers. We added metadata to track whether an implementation's output for an algorithm matches the output of the other implementations, as simple sanity check of each implementation's correctness.

The machine learning models trained by our tools are now stored in the database, including the metadata describing what results the model was trained on and the impact of individual features in the training data. We also added a notion of a "run configuration" which describes a single set of experiments on a specific dataset of graphs, using a specific version of the GPU code, and any other configuration. This way, we can accommodate results from different versions of the GPU code or different configurations

[5] After some small changes to fix manual mistakes in the data [91], even the 2-year-old data for [93] is still readable with the latest version of the tools.

in the same database, which allows us to more easily compare across variations in the configuration.

And even after 2.5 years of refinement we *still* found information missing from our schema in the process of writing this chapter. For example, we do not currently store information on: the operating system version, Linux kernel version, NVIDIA driver version, CUDA Software Development Kit (SDK) version, or other hardware information besides the GPU. We also lack a Digital Object Identifier (DOI) or Findability, Accessibility, Interoperability, and Reusability (FAIR) identifier for our input datasets, as the datasets we used did not have any — a result of their use not being widespread within computer science, yet.

We did not initially consider this information, as it was not very important for our efforts. All our experiments are run on the same cluster, so this information stays constant across our experiments. However, this information is important for the provenance of our data when we consider other researchers using our data.

3.2.4 Data Format Design Takeaways

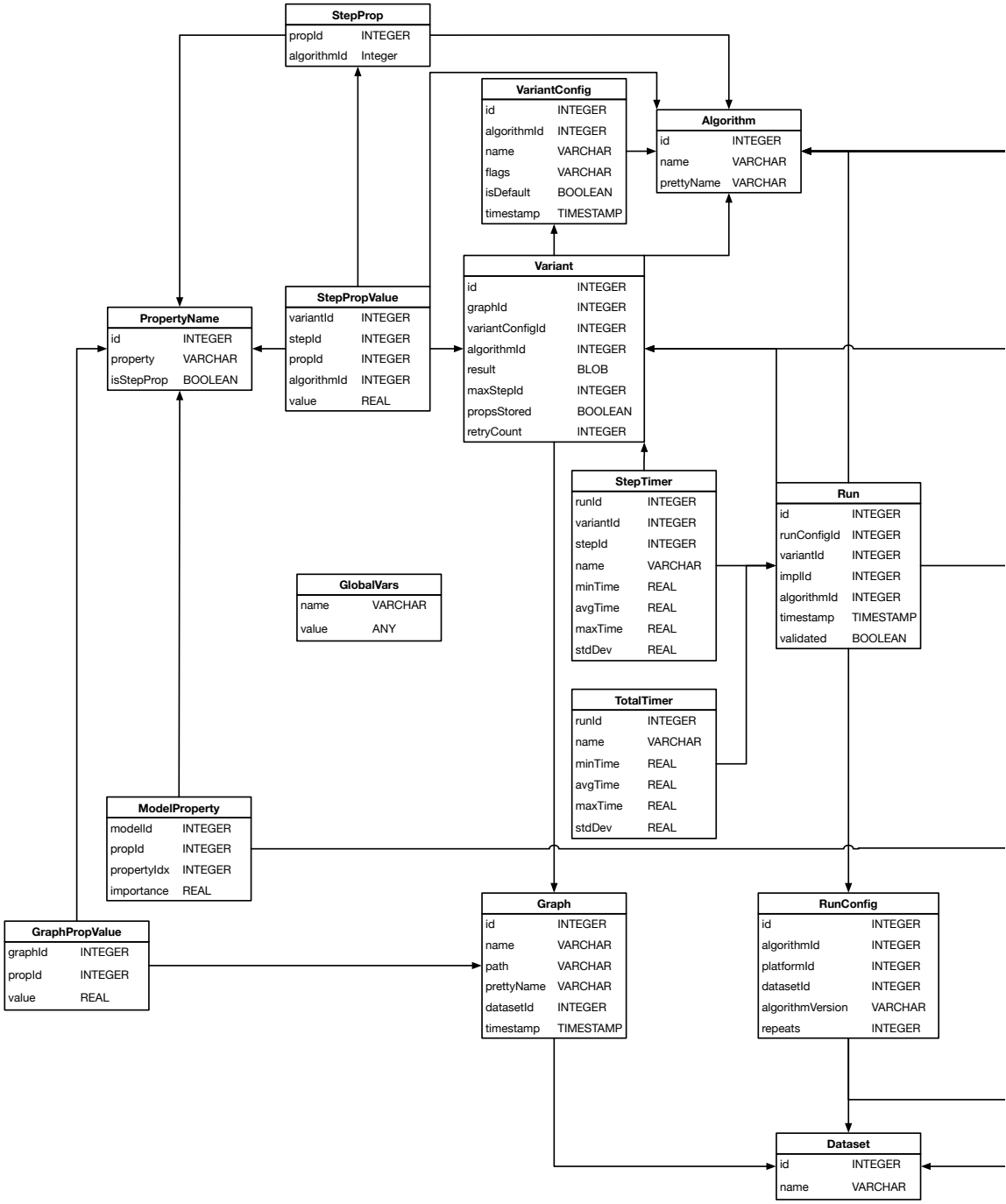
The switch to using a single database for all results, provenance, and analyses — together with the tools this allowed us to build — has been invaluable. It allowed us to explore and investigate larger sets of results more thoroughly and in a fraction of the time it would have taken without it[6].

At the same time, this single database has resulted in making our data more accessible, transparent, and available for others who wish to explore our work or experiment with their own analyses on the data we gathered. There are two main takeaways from our experience working on this data format.

First, defining a data format and building tooling around that should, in hindsight, have been central in this research from the start, because it is the increased scale of data and experiments that enables us to investigate hypotheses that we could not handle before.

Second, it is crucial to consider how we accommodate changes, extensions, and additions to the data and metadata we store in our data formats. No matter how systematically we thought through the data we needed, we kept finding things we missed or did not realise. The continuous evolution and refinement of our tools and data format has been invaluable for our investigations. Versioning our data's schema and investing the engineering time to support migration from old data allows us to keep evolving and refining our tooling without throwing out all previous experimental work.

[6] This estimation is based on our own experience with both solutions.


StepProp
propId INTEGER
algorithmId Integer
VariantConfig
id INTEGER
algorithmId INTEGER
name VARCHAR
flags VARCHAR
isDefault BOOLEAN
timestamp TIMESTAMP
Algorithm
id INTEGER
name VARCHAR
prettyName VARCHAR
Variant
id INTEGER
graphId INTEGER
variantConfigId INTEGER
algorithmId INTEGER
result BLOB
maxStepId INTEGER
propsStored BOOLEAN
retryCount INTEGER
PropertyName
id INTEGER
property VARCHAR
isStepProp BOOLEAN
StepPropValue
variantId INTEGER
stepId INTEGER
propId INTEGER
algorithmId INTEGER
value REAL
StepTimer
runId INTEGER
variantId INTEGER
stepId INTEGER
name VARCHAR
minTime REAL
avgTime REAL
maxTime REAL
stdDev REAL
Run
id INTEGER
runConfigId INTEGER
variantId INTEGER
implId INTEGER
algorithmId INTEGER
timestamp TIMESTAMP
validated BOOLEAN
GlobalVars
name VARCHAR
value ANY
TotalTimer
runId INTEGER
name VARCHAR
minTime REAL
avgTime REAL
maxTime REAL
stdDev REAL
ModelProperty
modelId INTEGER
propId INTEGER
propertyIdx INTEGER
importance REAL
Graph
id INTEGER
name VARCHAR
path VARCHAR
prettyName VARCHAR
datasetId INTEGER
timestamp TIMESTAMP
RunConfig
id INTEGER
algorithmId INTEGER
platformId INTEGER
datasetId INTEGER
algorithmVersion VARCHAR
repeats INTEGER
GraphPropValue
graphId INTEGER
propId INTEGER
value REAL
Dataset
id INTEGER
name VARCHAR

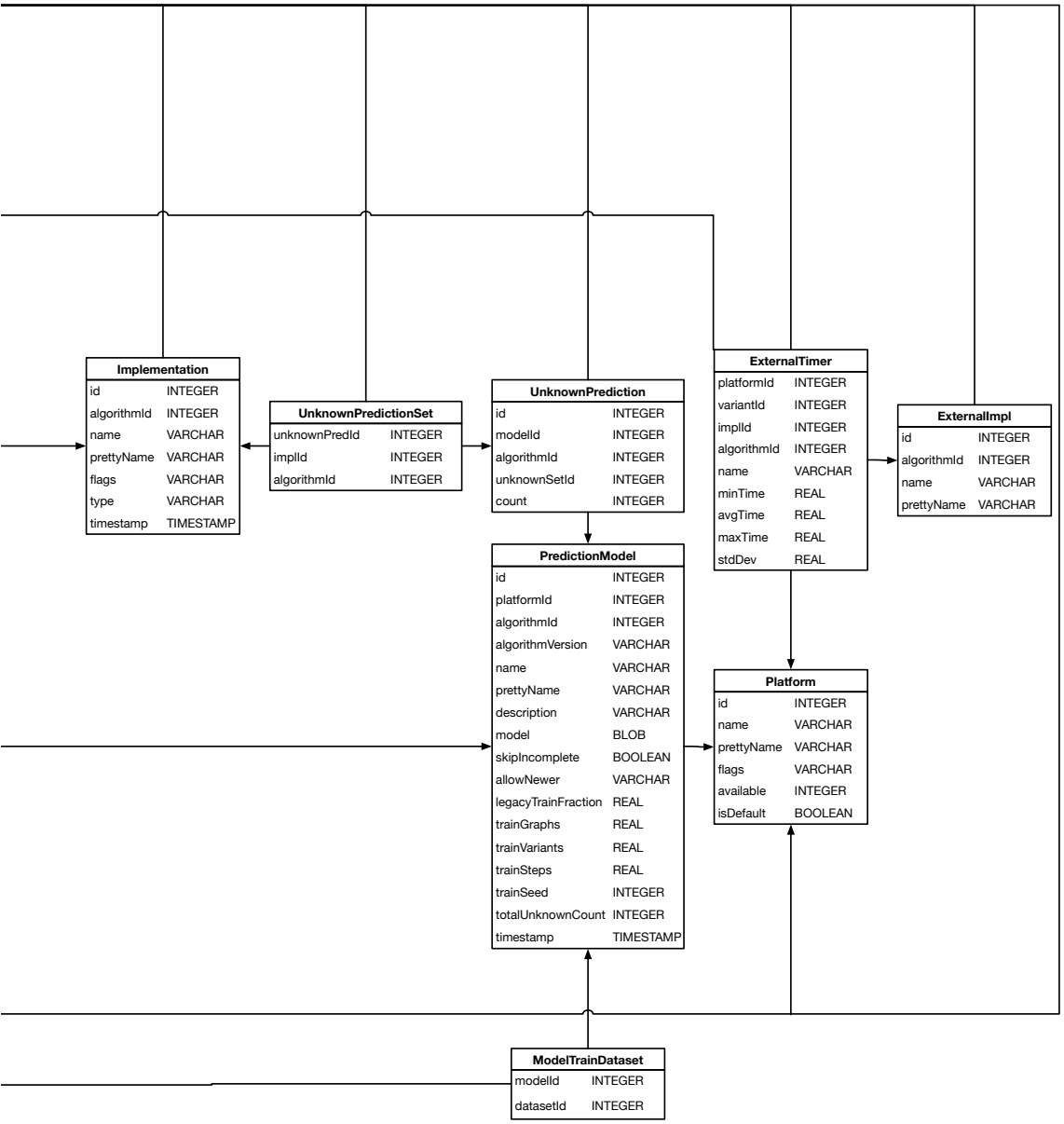


Figure 3.3: Schema for the current version of the data format. Explained in Section 3.3 and Appendix A

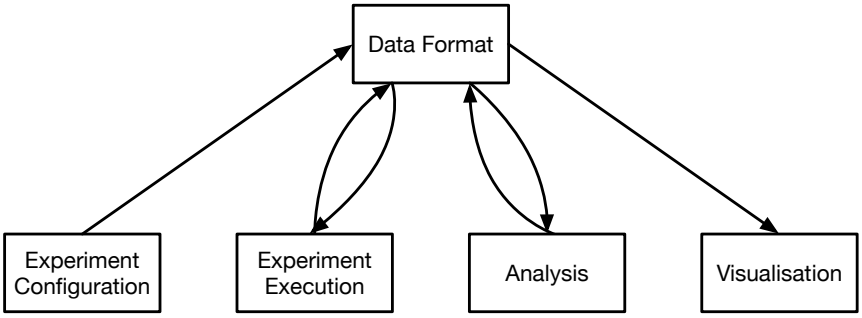


Figure 3.4: Stages of interaction with our data format.

3.3 High-level Implementation & Tooling

Our data format is the hub through which all our tools interact with the data and each other. Figure 3.4 illustrates the 4 conceptual modes of interaction with our data format:

1. Defining configurations of experiments to be run,
2. running the configured experiments,
3. analysing results of past runs, and
4. plotting and visualising gathered data and analyses.

These 4 conceptual modes roughly correspond to the 4 main executables in our codebase [89]. In this section we cover how these 4 modes of interaction are implemented in our tools and how the tools make use of our data format and the high-level design of our data format. For a detailed, technical description of the data format we refer to the explanation in Appendix A on page 137.

3.3.1 Experiment Configuration

We have to define what experiments to perform before we can start running them or analysing their results. Figure 3.5 on the facing page shows a simplified schema of the relevant information for defining experiments. This part of our data format serves two purposes:

1. It provides provenance information for our measurements, and
2. determines which experiments to run (see Section 3.3.2 on page 32).

As shown in Fig. 3.5 on the facing page a "run configuration" — i.e., a set of experiments — is defined by: a hardware platform, a set of graphs, and an algorithm.

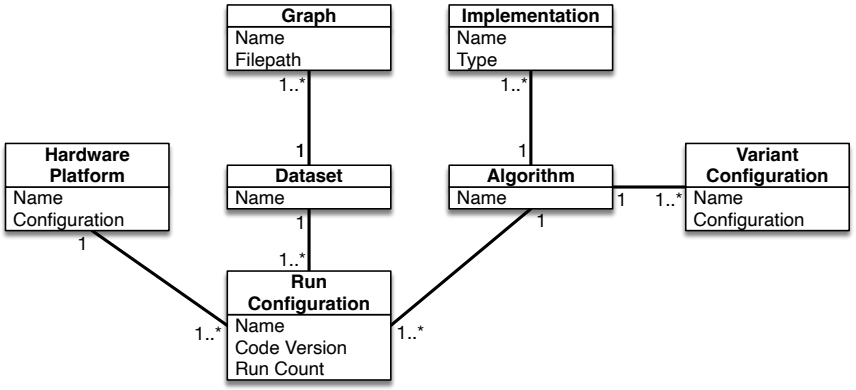


Figure 3.5: Simplified schema for the specification of runs of experiments.

In this thesis the hardware platform refers to the GPU used for the experiments. However, neither our data format nor our tools are limited to GPUs, other accelerators or Central Processing Units (CPUs) will work just as well.

The input data is made up of a named dataset of 1 or more graphs. Graphs are stored as a name and corresponding filesystem path.

Converting graphs to the right in-memory representation is slow, so we invented our own file format to speed this process up and assume that graphs are already in this format. The `normalise-graph` program can convert to our file format, it supports the common textual edge list format used by SNAP [56] and the Matrix Market Exchange format [10].

The algorithm indirectly specifies two other aspects of our experiments: the set of implementations and variants. For each algorithm we store a set of implementations of that algorithm. Each implementation representing a different parallelisation strategy and/or implementation approach of that algorithm, as discussed in Section 2.3 on page 14.

We distinguish two types of implementations: simple implementations and derived implementations. *Simple implementations* are standalone implementations of an algorithm. *Derived implementations* are made up of multiple simple implementations, plus the logic for switching between them at run time. The distinction between simple and derived algorithms will be covered in Chapter 7 on page 99.

Some algorithms can be run on the same graph in many ways. Consider, for example, BFS and a graph of N vertices; there are N unique BFS traversals of this graph, determined by the root vertex we select.

Variants identify algorithm specific configurations for an experiment, such as different root nodes. This allows us to consider the results of these different configurations separately.

The `Ingest` tool we built handles all the relevant operations for storing the information from Fig. 3.5 on the previous page in our database. It provides a number of subcommands for operating on our data format. The most important of these commands are:

`add`, registers new data in the database.

`list`, prints an overview of stored data.

`query`, prints detailed information about a single entry.

The `list` and `query` commands have their own subcommands to specify which table or entry to print information. These subcommands come with their own command-line help, clarifying what is printed by them. Registering new experiment configurations is handled via the subcommands of `add`:

`add platform`
: Registers a new supported hardware platform.

`add graphs`
: Registers new input graphs.

`add algorithm`
: Registers a new algorithm.

`add implementation`
: Registers a new implementation for an algorithm.

`add variant`
: Registers a new variant for an algorithm and graph pair.

`add run-config`
: Registers a new set of experiments to run.

These subcommands correspond to the schema in Fig. 3.5 on the preceding page, except for the missing "dataset" command, which is an implicit part of the command for adding new graphs to the database.

3.3.2 Experiment Execution

With our experiment configurations defined in the previous section, we need to be able to actually run the experiments we define. There are two distinct types of data that we want to gather: timings and properties.

Timings are fairly self-explanatory; to say anything about performance we need to measure how long experiments take to run. The properties are important because the goal of our research is to look into the link

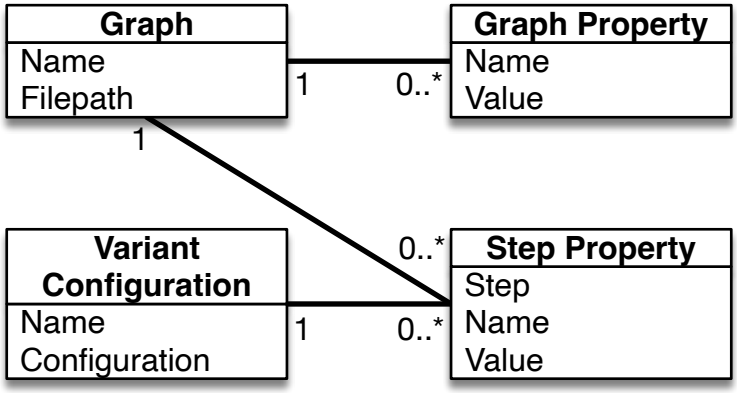


Figure 3.6: Simplified schema for the collected properties.

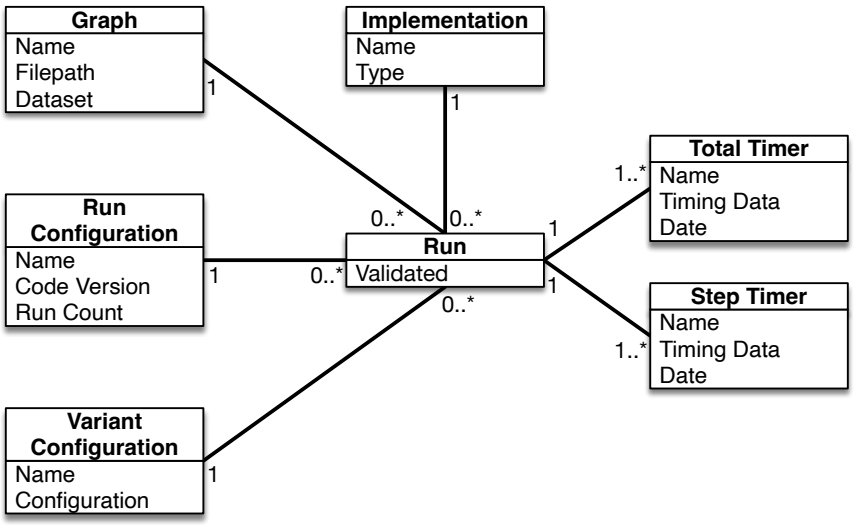


Figure 3.7: Simplified schema for the collected timings.

between the (structural) properties of graphs and the performance of our implementations.

As mentioned in Section 2.3 on page 14, we assume a Bulk Synchronous Parallel (BSP) model for our algorithms and implementations. In a BSP model, an algorithm consists of a number of supersteps with barrier synchronisations in between and maps well to GPGPU kernel invocations.

Figure 3.6 and Fig. 3.7 show simplified schemas for the collected properties and collected timings respectively.

We distinguish two types of properties in our data model: graph properties and step properties. Graph properties refer structural properties of our input graphs, such as the degree distribution, size, and diameter.

Step properties are algorithm specific run time properties, these are specific to algorithms and variants. An example is the frontier size for a BFS traversal. The frontier size is different for every superstep and individual variants — i.e., runs with different starting vertices — will have different values too.

Timing measurements store the wall-clock time taken by computations. We store *at least* the total execution time for each superstep and each complete run. Our data format accommodates an unlimited number of additional timers per superstep and per complete run to accommodate more fine-grained timing data.

The timings are stored as aggregate over multiple executions, as specified by the run configuration. Specifically, we store the minimum, average, and maximum observed timings, as well as the standard deviation of the timings.

3.3.2.1 `kernel-runner`

The core of our execution stage is implemented by the C++ program `kernel-runner`. This executable provides a unified interface for different graph algorithms and GPUs. `kernel-runner` supports two modes of operation: interactive and batch.

In interactive use, it can list details about supported algorithms and implementations, query information about available GPUs, and run individual implementations on graph files. In batch mode, the program reads jobs from `stdin`, and reports job completion to `stdout`. The algorithm's results and timings logged to job-specific output files.

The `kernel-runner` program provides an API for registering algorithms and their implementations. On start-up `kernel-runner` scans specified directories for dynamic libraries implementing this API and loads all the libraries it finds. This lets us speed up (re)compilation and keep algorithm implementation separate from the other functionality.

An important part of the above API is a number of C++ templates and classes that enable the implementation of new algorithms with minimal effort. Sharing as much functionality as possible across algorithms and implementations.

The `Config` templates provided in `TemplateConfig.hpp` allow us to write a single implementation of an algorithm's host code that can be used with each GPU implementation of the algorithm. This includes loading the input graph from disk, converting it to the input format for our GPU kernel, transferring the graph to the device, and setting up the GPU kernels.

Having only a single host implementation shared across all device implementations means any instrumentation can be done in a single place, and shared across our device implementations. This includes logging for structural properties of graphs, logging the algorithm specific properties, and all the timers.

3.3.2.2 `Ingest`

The `kernel-runner` above gives us a consistent interface for running our different graph algorithms, but it does not integrate with our data format. This functionality is provided by the `run-benchmarks` subcommand of `Ingest`. This subcommand queries which experiment configurations do not have corresponding properties or timings stored in the database and proceeds to run any missing experiments.

The `Ingest run-benchmarks` command spawns multiple parallel instances of the `kernel-runner` in batch mode and distributes the missing benchmark jobs across these processes. By default, this command uses the Slurm Workload Manager [47] as this is the default scheduler of the Distributed ASCI Supercomputer 5 (DAS5) [7]. However, this default can be overridden with `Ingest set run-command` command which allows users to specify a custom command for launching jobs using their own cluster management software or queueing system.

We also store a hash of the algorithms result for each variant, allowing us to check that all of our implementations correctly produce the same result. This does not work for all algorithms, since the results of some algorithms — e.g., PageRank — are affected by the fact that IEEE-754 floating-point operations are not commutative.

These floating-point values can only be checked by doing a pairwise, epsilon-based comparison. Storing the full result for every experiment is not feasible, as the size can be several gigabytes. Instead, we provide a separate `Ingest validate` command. This command repeats every run whose result hash is incorrect and performs a full pairwise comparison of floating-point values to see if the results match when using an epsilon-based comparison.

3.3.3 Analysis

Figure 3.8 on the following page shows a simplified schema of the inputs we train our models on. In this section we limit ourselves to discussing the high-level operations provided by our tools and the inputs we store to train models and perform analyses on. For the gory details of the exact model metadata we store we refer back to Appendix A on page 137. For details on the training method, model metadata, and model evaluation we refer to Chapters 7 and 8 on page 99 and on page 111.

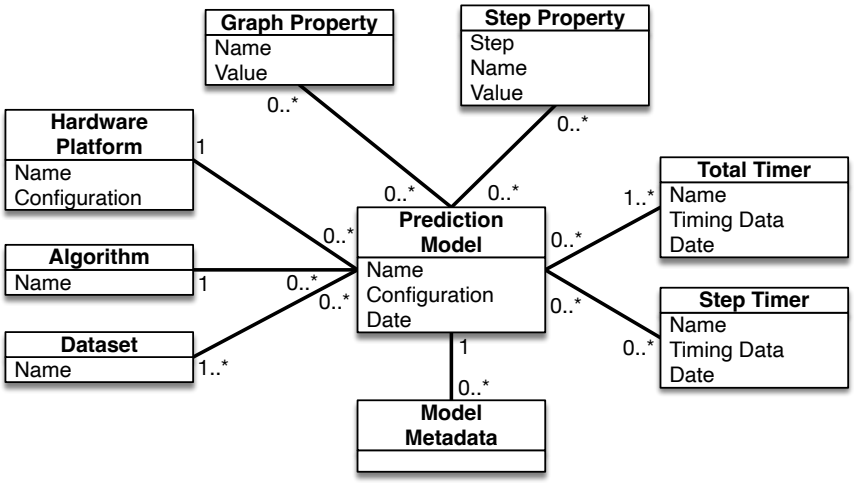


Figure 3.8: Simplified schema for a model's training inputs.

We define a "training configuration", as shown in Fig. 3.8, to include:

- The hardware platform our timings come from,
- the algorithm,
- one or more datasets of graphs,
- a set of graph properties,
- a set of step properties of the algorithm,
- the step timings of the algorithm,
- the total times for each algorithm,

Many machine learning models require separate training and validation sets, so our queries support splitting the above data into separate training and validation sets. We support uniform random splits based on graphs, specific variants, and individual BSP steps. This randomisation is based on a seeded Pseudo Random Number Generator (PRNG).

We store the PRNG seed and chosen training/validation split with the model. This allows us to rerun our experiments with the exact same training/validation sets later, and also gives others clear provenance of the exact data a model was trained on.

The `Model` executable has subcommands for all our model related interactions with our data format. The most relevant subcommands are:

`list`
: Lists all models stored in the database.

`query`
: Prints detailed training configuration and metadata for a model.

`train`
: Train a new model from a given configuration and store the model and its metadata in the database.

`validate`
: Prints detailed statistics about a prediction model's accuracy on both the validation and total data.

`evaluate`
: Prints a detailed performance comparison between 1 or more models and previously measured implementations. Comparison is done for 1 specific GPU and either all known graphs or across 1 or more specific datasets of graphs.

`compare`
: Prints detailed performance comparison of measured implementations. Comparison is done for 1 specific GPU and either all known graphs or across 1 or more specific datasets of graphs.

`show`
: Prints the predicted implementation and (optionally) properties for each BSP step for a specific variant.

`export/export-source`
: Export a trained model as C++ library or source file that can be plugged into our `kernel-runner`.

In addition to printing various detailed summaries the `Model` executable can also be used to directly produce LaTeX tables, such as the ones in Chapters 7 and 8 on page 99 and on page 111. This lets us generate all the graphs and tables in this thesis directly from our dataset, ensuring the accuracy every plot and table.

3.3.4 `Plot`

With all this data conveniently gathered into a single database, we still need a way to visualise it all. The `Plot` executable handles this. There are three separate subcommands: `report`, `bar` and `heatmap`.

The `report` helps identify which variants may be interesting to plot. Its output includes the variants where each implementation performs the best/worst and variants where there measured timings are "statistically interesting". Examples of "statistically interesting" properties are:

- Smallest/largest difference between best and worst implementation,

- smallest/largest spread of timings,
- smallest/largest skewness, and
- smallest/largest kurtosis.

The `bar levels` subcommand plots the run times of a set of implementation on the specified GPU against the different BSP supersteps of a specific variant. Allowing us to compare how the performance of individual implementations changes across a run on a single input graph.

The `bar totals` subcommand plots the total run times for a set of implementations on given set of graphs. And finally, the `bar vs-optimal` plots the run times of a set of implementations normalised to the "optimal" performance according to our results.

The `heatmap total` and `heatmap levels` subcommands correspond to their `bar` counterparts, but visualises the data using a heatmap. The `heatmap predict` subcommand extends the output of the `heatmap total` command to include 1 or more predictors.

3.4 Lessons Learned

In other empirical sciences the engineering work going into experimental setups[7] is treated as a scientific contribution in its own right. There are journals dedicated to instrumentation and measurements. This is in stark contrast with computer science, where software engineering and programming work is often viewed as distinct from the "real computer science" and rarely as a scientific contribution.

Perhaps this is an artefact of the *same* people building the "instrumentation" and doing the experiments. Or maybe it is due to computer science rarely having *physical* experimental setups. Regardless of the reasons, this difference is holding empirical computer science back. The quality of the software engineering determines how easy experiments are to reproduce and validate; and, in any empirical science the quality of your results is only as good as your ability to reproduce your results. An aspect that is, unfortunately, still often overlooked in computer science [23].

The experiments within this thesis, despite only addressing a tiny part of graph processing on GPUs, involve 56 implementations of BFS, 19 implementations of PageRank, 247 graphs from the KONECT dataset, and 109 graphs from the SNAP dataset. Benchmarking all these possible combinations leads to:

$$(56 + 19) \times (247 + 109) = 26,700 \text{ experiments}$$

[7] Although other empirical sciences would probably refer to it "instrumentation", rather than engineering work.

The above 26,700 experiments is *before* we even consider repeating these experiments across multiple GPU generations or the fact that BFS has different behaviour for every possible starting vertex in a graph. At this scale, reproducibility becomes wishful thinking without adequate tooling and data management.

As such, it is our opinion this software pipeline is an integral part of the scientific contributions in this thesis. It forms the backbone of the methodology we use in this thesis and allows other to do similar investigations on other graph algorithms, datasets, hardware, and/or implementations.

There are additional benefits to using a single toolchain and SQLite database to pack and manage all this data and metadata, besides the reproducibility aspect. Having all the data about experiment configuration, results, and analysis configuration in a single database and toolchain makes tracking the provenance of results much simpler.

Furthermore, the single file format and wide support for SQLite make it trivial to share entire result sets with other researchers, letting them build their own work on top of the existing results without having to redo all the time consuming benchmarks themselves. The interested reader can find all the data that went into this thesis archived on Zenodo [90].

CHAPTER 4

Quantifying Performance Impact

The impact of input data on the performance of irregular algorithms is well-known in High-Performance Computing (HPC) and General Processing on GPU (GPGPU) communities; as is the fact that this impact varies across different implementations of the same algorithm. Graph processing, being the literal textbook example for irregular algorithm, is no exception.

In 2007, Lumsdaine et al. [61] argued that this irregularity is one of the main challenges to overcome on the road to high-performance graph processing. Many graph processing systems were developed since then [17, 37, 44, 45, 49, 66, 69, 97, 102]; however, the impact of input dependence and irregularity is rarely covered in the discussion of implementation techniques used in these systems.

The literature focuses almost exclusively on the efficiency with which implementations can be mapped to the underlying hardware; or other benefits, such as allowing higher-level optimisations or ease of programming.

This chapter is based on work previously presented in:
Merijn Verstraaten et al. "Quantifying the Performance Impact of Graph Structure on Neighbour Iteration Strategies for PageRank". In: *"Euro-Par 2015: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2015, pp. 528–540. ISBN: "978-3-319-27308-2"
Merijn Verstraaten et al. *Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach*. 2017. eprint: arXiv:1708.01159
Merijn Verstraaten et al. "Mix-and-Match: A Model-driven Runtime Optimisation Strategy for BFS on GPUs". In: *Proceedings of the 8th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE. 2018, pp. 53–60

The reason for this omission is straightforward. There has been very little systematic investigation into the link between graph structure and the performance of different implementation techniques. The large number of algorithms, techniques, and graphs makes it seem infeasible to perform a systematic investigation of their interplay and quantify their impact on performance.

By limiting ourselves to a single implementation technique and leveraging the further restrictions imposed by the GPGPU programming model, we reduce this seemingly infeasible problem space to a more manageable size.

In this chapter we cover the neighbour iteration primitive; discuss the possible methods for parallelising this primitive on the Graphical Processing Unit (GPU); show how to implement Breadth-First Search (BFS) and PageRank [73] using neighbour iteration; and show how different implementations of neighbour iteration affect the performance of these algorithms across graphs.

4.1 Neighbour Iteration Primitive

There are numerous graph algorithms and people keep inventing new ones and new variations of existing ones. As a result, it is clearly intractable to investigate the impact of graph structure for each individual algorithm. To have any hope of success, we need to ensure that our investigation is applicable to many algorithms.

One common method for problems like this is to identify a set of primitive operations that occur across different algorithms. If we can identify a small set of primitives that occur often and make up the bulk of each algorithm, we can then limit our investigation to these primitives.

Attempts to define such a set of graph processing primitives have been made from several fields. In 2010, Buluç defined a set of primitives based on linear algebra [13]. Hong et al. created Green-Marl, a Domain Specific Language (DSL) for graph algorithms [43].

Other graph processing work did not set out to define a set of primitive operations, but ended up doing so implicitly as a result of limitations of their implementation.

Google's Pregel [63], and the Pregel-inspired Giraph [33], advocate a "think like a vertex"-model. This model requires vertex-centric code, where the only operations are updating vertex state and sending messages to neighbouring vertices.

Similarly, GraphLab [60] defines fold, merge, and apply primitives that let user code update vertex state, aggregate information across multiple vertices, and control the parallel execution of these.

While there is some overlap in the primitives defined by these various approaches, there is no consensus on what the best or most complete set of primitives for graph processing is or what the right level of abstraction is.

For example, in HPC communities BFS is commonly treated as an algorithm, whereas the model checking community thinks of BFS as a step or primitive in a larger model checking algorithm.

We limit ourselves to neighbour iteration, as it is the most common operation across the above frameworks. Almost all graph processing frameworks include it as either an explicitly built-in operation or implicitly as a trivial application of the implemented operations.

Neighbour iteration, as the name implies, is a primitive that applies an operation to each of a vertex' neighbours, doing this for all vertices in the graph.

In Section 4.2 we show different ways we can parallelise neighbour iteration on the GPUs, and in Sections 4.4 and 4.5 we show how to use it to implement PageRank and BFS.

4.2 Parallelisation Strategies for Neighbour Iteration

Limiting our investigation to neighbour iteration already reduces our problem space considerably, but the programming model and hardware constraints of GPUs let us reduce it even further.

In Section 2.3 on page 14 we observed that the key to effective graph processing on GPUs is to maximise the use of memory bandwidth. Maximising the use of memory bandwidth requires us to saturate the GPU with enough parallel work to maximise latency hiding.

We concluded that, given these restrictions, there are two "main" parallelisation strategies: vertex-centric (i.e., one thread per vertex) and edge-centric (i.e., one thread per edge). In this section we cover how these two parallelisation strategies can be applied to neighbour iteration, resulting in 6 main implementations of neighbour iteration.

4.2.1 Edge List & Reverse Edge List

These edge-centric implementations launch *one Compute Unified Device Architecture (CUDA) thread per edge*. Each thread is assigned an edge $(v, w) \in E$ to operate on. Threads read information from either the destination or origin vertex and process it. The common case is reading data from the origin and propagating it to the destination.

Edge list based implementations use the outgoing edges of every vertex, whereas reverse edge list based implementations use the incoming edges of every vertex. This difference affects the amount of memory coalescing and the access patterns exhibited at run time.

The advantage of these edge-centric parallelisation strategies is that they never suffer from workload imbalance, every thread in a warp performs the same amount of work. Edges with the same origin vertex are likely to end up in the same warp, which helps with coalescing memory accesses. The downside of this is that parallel updates result in highly contested atomic updates.

4.2.2 Vertex Push & Vertex Pull

These vertex-centric implementations launch *one CUDA thread per vertex*. Each thread is assigned a vertex and a list of neighbours to operate on. The *push* and *pull* terminology originates from work on undirected graphs.

For undirected graphs there is no difference between the incoming and outgoing edges of a vertex, as a result there are two common ways to process the neighbours of a vertex. A thread can read data from its assigned vertex, iterate over the neighbour list, and *push* that data out to each neighbour. Or a thread can iterate over the neighbour list, *pull* data from its neighbour, and update the data of its assigned vertex.

Since we assume directed graphs in this thesis, we can no longer use the same data representation for both these versions. Push implementations operate on the *outgoing* edges of a vertex, whereas pull implementations operate on the *incoming* edges. This difference in input data mirrors the difference between edge lists and reverse edge lists.

Both types of implementation are susceptible to workload imbalance — and thus performance loss — if vertices with wildly different degrees are grouped in the same warp. Push implementations, similar to edge-centric implementations, generate a lot of concurrent updates, requiring many atomic operations. However, they avoid the many redundant reads performed by edge list implementations.

Pull implementations require no atomic operations as there is only a single thread updating the data of each vertex. However, care needs to be taken that the data read from neighbours is not data that the neighbours are updating.

4.2.3 Virtual Warp Push & Pull

Virtual warps are a technique proposed by Hong et al. [45] to reduce the workload imbalance created by vertex-centric GPU implementations.

The CUDA warps are divided into smaller "virtual warps" of N threads. Each of these virtual warps is assigned N vertices to process. Instead of having every thread process a single vertex, like the previous section, all N threads in the virtual warp process the neighbours of a single vertex in parallel. Then all N threads process the next vertex, and so on until all N vertices assigned to the virtual warp have been processed.

Implementation	Graph 1	Graph 2	Average
A	100	300	200
B	150	150	150
C	300	100	200

Table 4.1: An example of the difficulty of comparing run times across multiple inputs.

This reduces the amount of load imbalance occurring within a virtual warp, since the workload of a virtual warp is spread out equally across that virtual warp. However, determining the optimal virtual warp size is challenging.

Moreover, different graphs can have different optimal virtual warp sizes. So instead of 2 implementations virtual warp push and virtual warp pull actually represent a family of implementations with different virtual warp sizes.

4.3 Intermezzo: Comparing Implementations

The implementation sections of this chapter, as well as the rest of this thesis, have us comparing different implementations of the same algorithms. However, the data dependence of these implementations makes meaningful comparison hard. In this intermezzo we explain the methodology and terminology we use to compare implementations in this chapter and the rest of this thesis.

Comparing the performance of implementations on a single graph is straightforward, simply compare the run times and done. Once we want to talk about the performance of an implementation "in general", things become problematic.

Consider the example implementations, graphs, and run times shown in Table 4.1. The best implementation differs for each graph and implementation B is never the fastest, but has the lowest average time.

For a few graphs we can sidestep this issue and simply tabulate all the results in a single table, like our example. This is the approach taken by many current graph processing papers, but it is often unclear how representative their chosen graphs are and whether we can truly generalise from the shown results.

The experiments in this thesis involve hundreds or thousands of runs of our implementations. While the full results are publicly available [90], it is simply not feasible to tabulate them in a thesis. The pages are too small and human intuition and pattern recognition cannot handle numbers at that scale.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Edge List	28.47×	4.58×	50%	22%	3%	66.98×

Table 4.2: An example performance aggregate table.

We need one or more aggregate metrics that let us talk about the performance of an implementation over a set of data. To the best of our knowledge there are no standard metrics or representations for such aggregate performance data.

4.3.1 Aggregate Tables

Tables serve as our main method of displaying this aggregate performance data, listing per-implementation summaries of the aggregate performance over a set of variants[1]. Table 4.2 shows an example table. In the rest of this subsection we explain how to read these tables.

We start by defining an "optimal" time for every variant. This "optimal" time represents the fastest time possible for that variant, based on the times in our dataset. For PageRank this is equivalent to the fastest implementation for a variant. However, for BFS we have timings per "step" available. This allows us to take the fastest implementation at every BFS step of a variant and add their run times together to arrive at an "optimal" time that is better than merely the fastest overall implementation.

We can rank the performance of implementations by accumulating the run times of each implementation across all variants, getting their total time over all variants. If we divide the total time of each implementation by the total "optimal" time, we have a measure of the slowdown compared to the optimum possible over the entire dataset. We call this the *normalised total*, shown in the "Total" column of Table 4.2.

However, the normalised total alone is not enough. Deviations from the optimal on big graphs with long run times are punished disproportionately. A 10% deviation from optimal on a large graph can outweigh being the fastest implementation on many small graphs.

This disproportionate impact of large graphs is not a problem if determining the best performing implementation is all we care about. In that case, the performance on bigger/slower graphs should weigh more heavily than the performance on graphs that are fast to process anyway.

However, if we want to *understand* the performance of our implementations we need a more nuanced view. Such as an idea of how these deviations are spread across the variants. Are there only a few outliers with huge deviations from optimal? Are they many small deviations from optimal.

[1] A variant is one, algorithm specific, configuration for a graph.

To compare deviations from optimal across graphs of different sizes, we need to normalise the run times of each variant. We do this by taking the run time of each implementation and dividing it by the optimal time for that variant. The result is the slowdown compared to the optimal time. Consider the example in Table 4.1 on page 45. The normalised run time of C for graph 1 is 3, i.e., implementation C is 3× slower than the "optimal" implementation A.

These normalised run times give us a consistent indication of how much an implementation deviates from the optimum. We can accumulate the normalised run times for all implementations, divide by the number of variants, and compute the average deviation from optimal. We call this the *average error*, shown in the "Avg" column of Table 4.2 on the preceding page.

The average error alone does not give too much insight in how these errors are spread over the variants we consider. So, in addition to considering the average error across a dataset, we also compute for the percentage of times where an implementation is 1–2× of optimal, the percentage of times it is >5× of optimal, the percentage of times it is >20× of optimal, and the worst case in the entire dataset. These are shown in their respective columns of Table 4.2 on the facing page.

4.3.2 Measurement Accuracy

We use the aggregate tables, described in the previous section, throughout this thesis to compare the "overall" performance of various implementations. The validity of these aggregate comparisons is dependent on:

1. the accuracy of our measurements, and
2. absence of catastrophic error propagation.

Let us address the error propagation first. Our aggregate tables show implementation run times as a percentage or ratio of the optimal run time in our dataset. The run times of both the optimal and specific implementation are obtained by summing their respective mean run times across the entire dataset. As each measurement is independent, we assume their errors to be independent too. As a result, the error on the sum is bounded by the sum of the individual errors.

This means that the Relative Standard Error (RSE) of a summation is bounded by the largest RSE of its summands — if the largest RSE of all summands is $X\%$ then the sum of errors can never exceed $X\%$ of the sum.

The propagation of errors for division is:

$$RSE(\frac{X}{Y}) = \sqrt{RSE(X)^2 + RSE(Y)^2}$$

If we assume the same RSE X for the both the numerator and denominator, we get:

$$\sqrt{X^2 + X^2} = \sqrt{2 \cdot X^2} = \sqrt{2} \cdot \sqrt{X^2} \approx 1.4X$$

So, the upper bound on the RSE of our ratios is approximately 1.4× the RSE of our sums and the RSE of our sums is bounded by the RSE of our individual measurements. Ruling out any catastrophic error propagation from our measurements to our aggregates.

So, how accurate are our measurements? Our data [90] has timings of 995,931 full runs and 25,242,678 individual steps. The average RSE of the full runs is 0.6%, with only 7,845 (or 0.8%) of measurements having an RSE of over 5%. Similarly, the average RSE for the step timings is 0.5%, with only 51,750 (or 0.2%) of measurements having an RSE of over 5%.

Of course, averages only tell us so much. What we really care about are the outliers. What is the *maximum* measurement error in our results and how strongly do these impact our aggregates. At first glance we are in trouble. The maximum RSE is 99% for the full runs and 91% for the steps. With errors this big, our aggregates become meaningless.

Fortunately, all is not lost. Upon further investigation we discover that measurements with large RSEs fall largely into two categories:

1. Our dynamic implementations (see Chapter 7 on page 99), and
2. runs with short overall run times.

We will address the reason behind the large RSEs of our dynamic implementations in Chapter 7 on page 99, as well as the reason why these large errors do not impact out results.

If we disregard the measurements from our dynamic implementations, we are left with 3,045 (or 0.3%) of the full runs and 34,404 (or 0.1%) of the steps having a Relative Standard Error larger than 5%. These measurements are notable for their short run times, compared to the others.

The average run time of these 3,045 full runs is 0.24% of the overall average run time; and totalling up to only 0.0007% of the overall total run time. Similarly, the average run time of the 34,404 steps is only 2.8% of the average of overall step. In other words, the measurements with large RSEs are predominantly measurements that are too small to have significant impact on the overall result.

We computed, for every implementation, the fraction of the *total* run time that is made up by measurements with RSEs larger than 5%. The percentage of total run time made up by these "bad" measurements is, in the absolute worst case, 0.004% for the full runs and 0.02% for the individual steps.

We conclude that we can ignore the large Relative Standard Error on these measurements as they simply *cannot* meaningfully affect our overall

Id	Graph	# Vertices	# Edges	Dataset
1	actor-collaboration	382,219	30,076,200	KONECT
2	amazon0601	403,394	3,387,390	KONECT
3	flixster	2,523,390	15,837,600	KONECT
4	jester1	73,512	8,272,720	KONECT
5	patentcite	3,774,770	16,518,900	KONECT
6	wikipedia_link_en	12,151,000	378,142,000	KONECT
7	wiki_talk_ru	457,017	919,790	KONECT
8	higgs-social_network	456,626	14,855,800	SNAP
9	sx-stackoverflow-c2q	1,655,350	11,226,800	SNAP

Table 4.3: Selection of graphs to highlight interesting differences between implementations.

results. They make up such a tiny percentage of the overall run time that it would not meaningfully shift the ratios in our aggregate tables — even if our measurements were off by 10×.

4.3.3 Bar Plot Graph Selection

The aggregate tables from the previous section provide a good overview of the *aggregate* behaviour of implementations. However, sometimes we want to contrast individual runs and, as the popular adage goes, a picture is worth a thousand words.

For these comparisons we use bar plots with implementations grouped per variant to visualise how the run times differ between variants. We normalise the run times to ensure the differences between implementations on the same variant remain visible when plotting multiple variants in one graph.

For this normalisation we take the slowest, plotted, implementation on each variant and plot all run times on that variant as percentage of the slowest. Different variants will all have the same 0–100% scale, even if the actual run times differ by orders of magnitude, while preserving the relative performance of implementations for each variant.

We use a small set of graphs, shown in Table 4.3, to highlight interesting differences in the behaviour of implementations. We selected these graphs using the `Plot report` command described in Section 3.3.4 on page 37 to identify graphs/variants that exhibit statistically "interesting" behaviour.

Behaviours of interest include: Variants with either large or small differences between the best and worst implementation, skewness of the run times, or variants where implementations deviate from their "average" performance.

4.4 Implementing PageRank

PageRank is an algorithm for ranking vertices in a graph based on their *importance* [74]. Here, the importance of a vertex is defined as the chance of visiting that vertex while randomly walking the graph with a fixed chance of stopping and picking a new random starting point. PageRank was initially developed at Stanford, but is best known as the original algorithm upon which Google's search empire was built.

Given a graph $G = (V, E)$ the PageRank of a vertex $v \in V$ is given by the fixed point function PR defined in Eq. (4.1).

$$PR(v) = \frac{1-d}{|V|} + d \sum_{w \in N(v)} \frac{PR(w)}{\rho(w)} \tag{4.1}$$

The damping factor d represents the chance of continuing a random walk after each step, the degree function $\rho(w)$ gives the outgoing degree of a vertex $w \in V$. $N(v)$ denotes the neighbourhood of vertex $v \in V$, where the neighbourhood of a vertex is defined as:

$$w \in N(v) \iff (w, v) \in E$$

We can define the uniform random chance of abandoning a random walk and starting a new random walk from a random other vertex as:

$$\frac{1-d}{|V|}$$

The chance of randomly walking from vertex $w \in N(v)$ to vertex v is:

$$d\frac{PR(w)}{\rho(w)}$$

That is, the chance of randomly walking to w, divided by the number of destinations reachable from w, multiplied by the chance d of continuing our random walk at all.

We can approximate the PageRank fixed point via an iterative algorithm. First, we assign each vertex an initial PageRank. As PageRank is a fixed point, the actual initial value is irrelevant; it only affects how long it takes to converge to a result. The convention is to assign every vertex an initial PageRank of $\frac{1}{|V|}$.

After the initial PageRank assignments, we alternate two steps until convergence. In the first step we compute the incoming PageRank value for each incoming edge. The incoming PageRank value of an edge is the current PageRank value of its origin, divided by the outgoing degree of the origin.

In the second step we apply the damping factor to the total incoming PageRank value, update the PageRank value for each vertex, and finally check whether the result has converged.

We determine convergence by either: (1) Checking that every vertex has a PageRank difference below a fixed δ, or (2) checking that the sum of all PageRank differences is below a fixed δ.

4.4.1 Implementations

Our GPU implementation of PageRank mirrors the two-step iterative approximation described above. One kernel computes the new PageRank value for every vertex, a second kernel applies the damping factor, updates the PageRank values and checks for convergence against our δ.

In Section 4.2 on page 43 we covered the 6 possible parallelisation strategies for neighbour iteration. In Algorithms 1 to 5 on pages 51–53 we give pseudocode implementations that show how these strategies apply to the first step of PageRank.

Algorithm 1 Edge List Update & Reverse Edge List Update

function EdgeList(*edges*, *pageranks*, *new_pageranks*, *idx*)
do
  $origin \leftarrow edges[idx].origin$
  $dest \leftarrow edges[idx].destination$
  $outgoingRank \leftarrow 0$

  if $degree(origin) \neq 0$ **then**
    $outgoingRank \leftarrow \frac{pageranks[origin]}{degree(origin)}$
  atomically do
    $new\_pageranks[dest]$ += $outgoingRank$
end function

The edge-centric implementations use an array of edges (i.e., pairs of origin and destination vertices) and a degree array that stores the outgoing degrees of each vertex. Resulting in a space complexity of $2 \cdot |E| + |V|$.

The vertex centric implementations are based on a Compressed Sparse Row (CSR) representation. With CSR we store an edge array holding the destination of each edge and an offset array that indicates where each vertex' edges start in the edge array. This leads to a space usage of $|V| + 1 + |E|$.

For the vertex push and vertex push warp implementations the outgoing degree of a vertex v can be computed directly from the CSR, by taking the edge array offset of $v + 1$ and subtracting the offset of v. The vertex pull and vertex pull warp implementations use a separate degree

Algorithm 2 Vertex Push Update

```
function VertexPush(
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    outgoingRank ← 0

    if degree(idx) ≠ 0 then
        outgoingRank ← pageranks[idx] / degree(idx)
    for nbr ∈ vertices[idx].neighbours do
        atomically do
            new_pageranks[nbr] += outgoingRank
end function
```

Algorithm 3 Vertex Pull Update

```
function VertexPull(
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    newRank ← 0

    for nbr ∈ vertices[idx].neighbours do
        newRank += pageranks[nbr] / degree(nbr)

    new_pageranks[idx] ← newRank
end function
```

Algorithm 4 Vertex Push Warp Update

```
function VertexPushWarp(
    warp_size,
    vertex_chunk,
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    for v ∈ vertex_chunk do
        outgoingRank ← 0

        if degree(v) ≠ 0 then
            outgoingRank ← pageranks[v] / degree(v)
        num_nbr ← degree(v)

        for nbr from 0 to num_nbr by warp_size do
            atomically do
                new_pageranks[nbr] += outgoingRank
end function
```

Algorithm 5 Vertex Pull Warp Update

```
function VertexPullWarp(
    warp_size,
    vertex_chunk,
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    for v ∈ vertex_chunk do
        newRank ← 0
        num_nbr ← degree(v)

        for nbr from 0 to num_nbr by warp_size do
            newRank += pageranks[nbr] / degree(nbr)
        atomically do
            new_pageranks[v] += newRank
end function
```

array, like the edge-centric implementations, requiring an additional $|V|$ elements worth of space.

In the second step of PageRank we take all the newly computed incoming PageRank values, apply the damping factor, and update the δ of this PageRank iteration. The implementation of this second kernel, shown in Algorithm 6, can be shared by all the previous update kernels; a vertex centric parallelisation is the only one that makes sense for this operation.

Algorithm 6 Consolidate PageRanks

```
function ConsolidatePageRanks(
    pageranks,
    new_pageranks,
    idx
) do
    newRank ← (1−d)/|V| + (d · new_pagerank[idx])
    diff ← abs(newRank − pageranks[idx])

    updateDeltaDiff(diff)

    pageranks[idx] ← newRank
    new_pageranks[idx] ← 0
end function
```

Looking at the kernel for vertex pull and vertex pull warp, we observe that it is performing more work than strictly necessary. Computing the incoming rank from every neighbour means that vertices that share neighbours unnecessarily replicate work of dividing the rank. We could simply move this division into the consolidation kernel, performing this computation once per vertex. We show pseudocode for these variations of the kernels in Algorithms 7 to 9 on pages 55–56.

For our edge-centric implementations, we implemented both a Structure of Arrays (SoA) and an Array of Structures (AoS) implementation of the edge data. In Central Processing Unit (CPU) code, the choice between these two representations affects the ability to vectorise code, use Single Instruction, Multiple Data (SIMD), and influences cache behaviour. It is not clear whether the same trade-offs apply on the GPU. By implementing both we can compare the results across graphs and see if there's any significant benefit to either approach.

4.4.2 Results

For our PageRank experiments we used a damping factor of 0.85. We ran each implementation for 100 iterations, rather than running until conver-

Algorithm 7 Vertex Pull NoDiv Update

```
function VertexPullNoDiv(
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    newRank ← 0

    for nbr ∈ vertices[idx].neighbours do
        newRank += pageranks[nbr]
    new_pageranks[idx] ← newRank
end function
```

Algorithm 8 Vertex Pull Warp NoDiv Update

```
function VertexPullWarpNoDiv(
    warp_size,
    vertex_chunk,
    vertices,
    pageranks,
    new_pageranks,
    idx
) do
    for v ∈ vertex_chunk do
        newRank ← 0
        num_nbr ← degree(v)

        for nbr from 0 to num_nbr by warp_size do
            newRank += pageranks[nbr]
        atomically do
            new_pageranks[v] += newRank
end function
```

Algorithm 9 Consolidate PageRanks NoDiv

```
function ConsolidatePageRanksNoDiv(
    vertexDegrees,
    pageranks,
    new_pageranks,
    idx,
    notFinal
) do
    newRank ← (1−d)/|V| + (d · new_pagerank[idx])

    if notFinal then
        newRank ← newRank / vertexDegrees[idx]
    diff ← abs(newRank − pageranks[idx])
    updateDeltaDiff(diff)

    pagerank[idx] ← newRank
    new_pageranks[idx] ← 0
end function
```

gence. This is because the number of iterations needed to converge can be affected by the order of operations, as floating-point operations are not commutative. The results presented here consist of the time the PageRank computation took, averaged over 30 runs, excluding data transfers to and from the GPU.

Our original experiments [94] were performed on an NVIDIA K20 GPU using version 5.5 of the CUDA toolkit. The results presented in this chapter use an updated version of the code, fixing an important performance issue and adding several extra implementations.

The experiments shown in this thesis were done using an NVIDIA TitanX GPU and version 10.0 of the CUDA toolkit on the Distributed ASCI Supercomputer 5 (DAS5) [7] cluster. Both the code [89] and results [90] for this thesis are available, archived on Zenodo. The most recent version of the code is available at `https://github.com/merijn/Belewitte`.

We ran all 19 variations of the 7 PageRank implementation strategies shown above on the graphs from the SNAP [56] and KONECT [51] datasets. The aggregate performance across both datasets is shown in Table 4.4 on the facing page. For brevity's sake we include only the fastest virtual warp implementations.

In Fig. 4.1 on the next page we show the normalised run times for the graphs mentioned in Section 4.3.3 on page 49. This figure highlights the performance volatility of our implementations. The run times for graph 5 are all fairly close, whereas graph 7 shows vertex push being 1–2 orders

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Struct Edge List	1.15×	1.09×	98%	0%	0%	3.48×
Edge List	1.18×	1.16×	98%	0%	0%	3.54×
Vertex Pull NoDiv	2.29×	2.84×	55%	11%	0%	18.80×
Reverse Edge List	2.54×	2.15×	71%	8%	0%	10.60×
Reverse Struct Edge List	2.55×	2.14×	71%	8%	0%	10.76×
Vertex Push Warp 16–64	2.57×	4.05×	36%	17%	2%	66.63×
Vertex Pull Warp NoDiv 16–64	3.25×	5.32×	12%	38%	1%	27.46×
Vertex Pull Warp 16–64	4.49×	6.79×	5%	49%	3%	35.05×
Vertex Push	5.28×	12.20×	37%	34%	8%	642.08×
Vertex Pull	13.20×	16.14×	13%	60%	28%	143.45×

Table 4.4: Aggregate performance of our core PageRank implementations across graphs from KONECT and SNAP. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

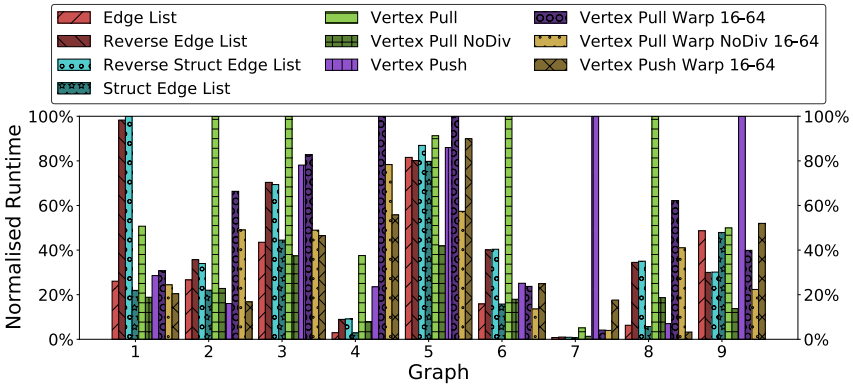


Figure 4.1: Normalised run times of our PageRank implementations for the graphs from Table 4.3 on page 49.

magnitude away from the other implementations.

We see the edge list implementations performing well on most of the 9 selected graphs. This follows our expectations from the results in Table 4.4, we see the two edge list implementations perform the best overall, both in terms of average error from the optimum and the total time.

At the same time we see that there is room for improvement. We see "Vertex Pull NoDiv" beating the edge list implementations by 2–3× on graphs 5 and 9. On graph 9 we also see the reverse versions improving on the edge list implementations.

4.5 Implementing Breadth-First Search

BFS is a traversal algorithm. It was first invented in 1943–1945 by Konrad Zuse to find connected components of a graph as part of his work on Plankalkül [38, 103], however his thesis on the subject was not published until 1972. In 1959, Moore independently reinvented BFS to find the shortest path out of a maze [68].

Other applications of BFS include: wire routing [55], copying garbage collection [22], single source shortest path in unweighted graphs, and maximum flow [32].

Algorithm 10 Breadth-First Traversal

```
frontier ← {v}

while frontier ≠ ∅ do
    newFrontier ← ∅

    for w ∈ frontier do
        for u ∈ N(w) do
            if NotVisited(u) then
                newFrontier ← newFrontier ∪ {u}

    frontier ← newFrontier
```

Algorithm 10 shows a pseudocode implementation of a BFS traversal. In this pseudocode $N(v)$ denotes the neighbourhood of vertex $v \in V$, where the neighbourhood of a vertex is defined as:

$$w \in N(v) \iff (v, w) \in E$$

They key distinction between the use of neighbour iteration in BFS and PageRank is the frontier. Where PageRank always iterates over the neighbours of *every* vertex, BFS only has to iterate over the neighbours of vertices in the frontier. We can use either a Gather-Apply-Scatter (GAS) or masking based approach to handle the frontier during neighbour iteration.

With a GAS-based approach we construct a temporary data structure for the frontier — the gather step. Then run our parallel neighbour iteration per the strategies described in Section 4.2 on page 43 — the application step. Finally, the results from the temporary data structure are written back to the original data structure — the scatter step.

The downside of a GAS approach is that the gather and scatter steps introduce extra overhead to copy data into and out of the temporary data structure, this overhead is linear in the size of the frontier. Additionally,

implementing an efficient queue or work list on the GPU is a non-trivial challenge due to the memory consistency model.

With the masking approach we keep the 1 thread per vertex or edge parallelism described in Section 4.2 on page 43, but wrap the code of each thread in a conditional check that determines if the thread belongs to the frontier.

The downside of a masking approach is that threads which are not in the frontier are idle for the duration of the entire computation, while taking up slots in thread blocks and the scheduler of the Streaming Multiprocessors (SMs). This reduces the SMs' efficiency with the same ratio as the ratio between frontier size and graph size.

In the rest of this section we restrict ourselves to the easier to implement masking variants of BFS.

4.5.1 Implementations

Our GPU implementation of BFS follows straightforwardly from the pseudocode in Algorithm 10 on the preceding page and parallelisation strategies described in Section 4.2 on page 43 and the start of this section.

We allocate an array of depths, that stores the distance between each vertex and our root vertex. We initialise the depth of the root vertex to 0 and all other vertices to infinity. The core loop of BFS consists of processing all nodes in the current frontier, updating the depths of their neighbours and the frontier, and then repeating until the frontier is empty.

Algorithms 11 to 15 on pages 60–61 show pseudocode implementations of the strategies from Section 4.2 on page 43 applied to BFS.

With our masking based approach, the first thing every thread does is check whether they are part of the frontier. The edge list implementations do this by comparing the depth of the edge's origin with the current depth, while the push and push warp implementations do this by comparing the depth of their assigned vertex with the current depth.

For the pull and pull warp implementations the *threads* in the frontier are not directly derived from the vertices in the BFS frontier. Instead, all threads corresponding to vertices that do not yet have a depth are part of the frontier, since only vertices without a depth can find a new depth by iterating over their neighbours.

At every BFS level, zero or more new vertices are discovered, forming the frontier for the next level. The size of this frontier needs to be tracked, since the algorithm terminates when no new vertices are discovered. We do this by having each thread track how many new vertices it discovers, and aggregating these counts at the end of each BFS level to compute the new frontier size.

We implemented four different aggregation variants for this frontier count. Our first variant uses a global variable, with every thread perform-

Algorithm 11 Edge List BFS & Reverse Edge List BFS

```
function EdgeList(edges, depths, current_depth, idx)
do
    origin ← edges[idx].origin
    dest ← edges[idx].destination

    if depths[origin] ≠ current_depth then
        return

    atomically do
        depths[dest] ← min(depths[dest], current_depth + 1)
end function
```

Algorithm 12 Vertex Push BFS

```
function VertexPush(vertices, depths, current_depth, idx)
do
    if depths[idx] ≠ current_depth then
        return

    for nbr ∈ vertices[idx].neighbours do
        atomically do
            depths[nbr] ← min(depths[nbr], current_depth + 1)
end function
```

Algorithm 13 Vertex Pull BFS

```
function VertexPull(vertices, depths, current_depth, idx)
do
    if depths[idx] <= current_depth then
        return

    for nbr ∈ vertices[idx].neighbours do
        if depths[nbr] = current_depth then
            depths[idx] ← current_depth + 1
end function
```

Algorithm 14 Vertex Push Warp BFS

```
function VertexPushWarp(
    warp_size,
    vertex_chunk,
    vertices,
    depths,
    current_depth,
    idx
) do
    for v ∈ vertex_chunk do
        if depths[v] ≠ current_depth then
            continue

        for nbr from 0 to num_nbr by warp_size do
            atomically do
                depths[nbr] ← min(depths[nbr], current_depth + 1)
end function
```

Algorithm 15 Vertex Pull Warp BFS

```
function VertexPullWarp(
    warp_size,
    vertex_chunk,
    vertices,
    depths,
    current_depth,
    idx
) do
    for v ∈ vertex_chunk do
        if depths[v] <= current_depth then
            continue

        for nbr from 0 to num_nbr by warp_size do
            atomically do
                depths[nbr] ← min(depths[nbr], current_depth + 1)
end function
```

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Vertex Push Warp 16–64	2.17×	3.53×	48%	9%	3%	82.26×
Edge List	3.97×	1.75×	83%	4%	0%	51.59×
Struct Edge List	4.77×	1.97×	81%	5%	0%	64.96×
Vertex Pull Warp 16–64	5.16×	9.88×	13%	29%	8%	1174.28×
Vertex Push	7.47×	18.56×	41%	37%	10%	1197.23×
Reverse Edge List	7.48×	2.67×	74%	9%	1%	93.75×
Reverse Struct Edge List	8.05×	2.83×	72%	12%	1%	108.14×
Vertex Pull	12.53×	25.23×	33%	37%	14%	2227.77×

Table 4.5: Aggregate performance of our core BFS implementations across graphs from KONECT and SNAP. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

ing atomic operations directly on this variable. Our second variant tries to alleviate the atomic operation penalty by batching the atomic operations performed by a single thread.

The literature suggests that the number of atomic operations and contention can be reduced further by performing a reduction within a warp or block [41] *before* performing the global atomic operations. Thus, our third and fourth variants perform a warp and a warp-and-block reduction, respectively, before updating the frontier size from only 1 thread per warp.

4.5.2 Results

Our original experiments [92, 93] were performed on an NVIDIA TitanX GPU using version 8.0 of the CUDA toolkit. As with PageRank, the results presented in this chapter use an updated version of the code, adding several extra implementations.

The new experiments shown in this thesis were done on an NVIDIA TitanX GPU and version 10.0 of the CUDA toolkit on the DAS5 [7] cluster. Both the code [89] and results [90] for this thesis are available, archived on Zenodo. The most recent version of the code is available at `https://github.com/merijn/Belewitte`.

Adding in the 4 different frontier aggregation variants and different virtual warp sizes, we end up with 56 different implementations based on the 6 implementation strategies shown above. We ran these 56 implementations on the graphs of both KONECT and SNAP averaging our timings over 30 runs. The aggregate performance across both datasets is shown in Table 4.5. For brevity's sake we include only one of the 4 frontier aggregation methods and only the fastest virtual warp implementation.

We immediately see that BFS behaves different from PageRank. The edge list implementations again have the smallest average error, but the

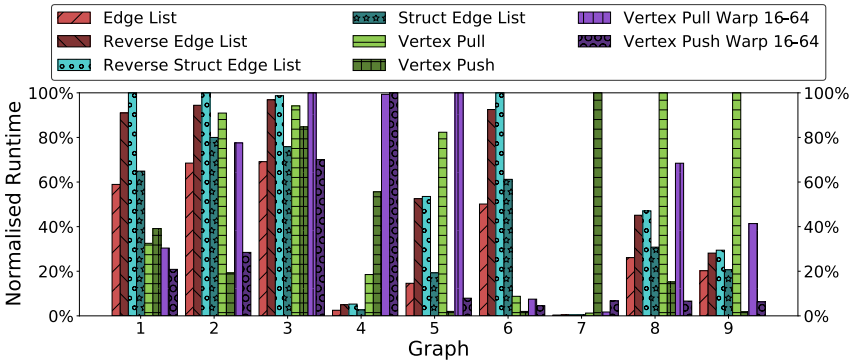


Figure 4.2: Normalised run times of our BFS implementations for the graphs from Table 4.3 on page 49.

error is several times bigger than it was for PageRank. Most notable, when comparing with the PageRank results, is that the edge lists implementations lost the top position to the virtual-warp-based push implementation.

The edge list implementation is within 1–2× of optimal 83% of the time and has an average error of only 1.75×. The virtual warp push implementation is within 1–2× only 48% of the time and has an average error of 3.53×. Yet, despite being worse in all these metrics it is almost twice as fast as the edge list implementation.

The most likely explanation for this behaviour is that large graphs that take long to process have a much bigger impact on the total run time. If graph A has an optimal time of 100 seconds and graph B has an optimal time of 100,000 seconds, as 10% error on graph B is going to impact the total more than a 10% error on graph A. These numbers indicate that the "Vertex Push Warp" implementation does significantly better on the large graphs than the "Edge List" implementation.

In Fig. 4.2 we show the normalised BFS run times on the graphs from Table 4.3 on page 49. We see that the BFS run times are even more volatile than PageRank. Where the PageRank edge list implementations were at the top for most graphs, here we see considerable variation. For graph 4 the edge list implementation beat the rest by 1–2 orders of magnitude. Whereas they are among the worst implementation for graph 3.

So far we assumed that the "optimal" time is simply the time of the fastest implementation, as we have with PageRank. However, the behaviour of BFS is not nearly as static as PageRank is. At every superstep there is a different frontier of active threads, with the run time of each implementation differing at every step of the BFS, sometimes by multiple orders of magnitude.

In Fig. 4.3 on the next page we have plotted the run times of our core

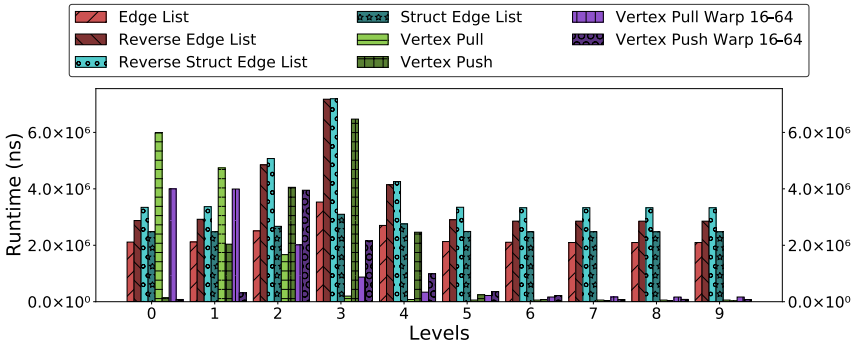


Figure 4.3: Run times of different BFS implementations per level of the `actor-collaborations` graph from KONECT.

BFS implementations across all the BFS levels on the actor-collaboration graph. Here we see "Vertex Pull" performing terribly in the first three levels of the BFS, but from level 3 on it is consistently faster than the other implementations.

We also see that the edge list implementation is substantially slower than the best implementation at each level. However, its more stable performance across levels lead to it seeming competitive when considering the entire graph.

This leads to a new "optimal" time, where we use the best possible implementation at every level of our BFS. In Table 4.6 on the facing page we compare the performance numbers of our implementations against this new optimal time. The new "Best Non-switching" entry refers to our old optimal — i.e., picking the best implementation for each graph and using it for all levels of the BFS.

The results in Table 4.6 on the next page and Fig. 4.4 on the facing page show that, even if we magically know the best implementation for each graph ahead of time, we are still leaving a considerable amount of performance on the table.

4.6 Summary

Our results demonstrate that performance of different implementations of a basic graph operation varies across different input graphs. This is shown by the significant fluctuations in run time and relative performance between our parallelisation strategies. These fluctuations appear in both our implementation of PageRank and BFS, and even within different levels of the same BFS traversal.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Best Non-switching	3.17×	2.32×	69%	8%	0%	37.82×
Vertex Push Warp 16–64	6.88×	8.20×	11%	29%	8%	234.05×
Edge List	12.60×	3.62×	54%	18%	1%	53.69×
Struct Edge List	15.15×	4.10×	50%	22%	2%	67.60×
Vertex Pull Warp 16–64	16.39×	17.71×	1%	68%	16%	1322.72×
Vertex Push	23.70×	39.87×	25%	53%	22%	1394.06×
Reverse Edge List	23.74×	5.87×	44%	28%	6%	97.57×
Reverse Struct Edge List	25.54×	6.19×	43%	30%	6%	112.54×
Vertex Pull	39.77×	39.48×	13%	58%	24%	2509.39×

Table 4.6: Aggregate performance of optimal BFS compared to our core BFS implementations across graphs from KONECT and SNAP. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

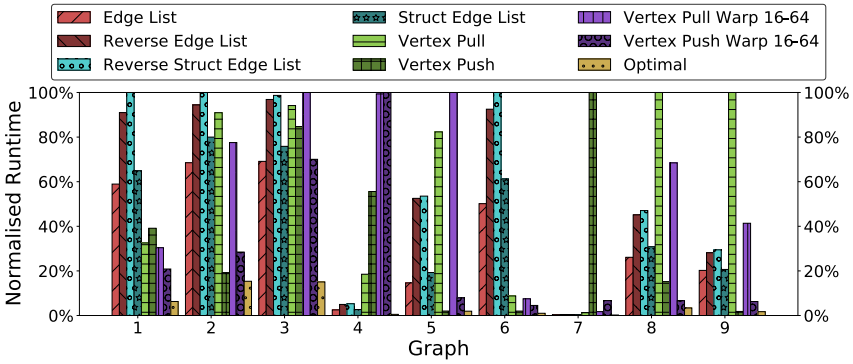


Figure 4.4: Normalised run times of our BFS implementations compared to "optimal" for the graphs from Table 4.3 on page 49.

One of our starting assumptions from the introduction is that the graph structure has a significant impact on performance. That is, we expect it to be (one of) the biggest influences on the observed performance.

In Sections 2.3 and 4.2 on page 14 and on page 43 we gave several examples of how the structure can affect the performance of different parallelisation strategies and our implementations. For example, the workload imbalance of vertex centric implementations depends on the degree distribution of vertices within a single warp, and thus the order in which vertices are stored in our CSR representation.

However, the results presented in this chapter are not sufficient to conclude that the shown performance differences can be attributed to graph

structure. Some of these performance differences may arise from a combination of the graph's structure and the actual in-memory representation of the graph. Such as the amount of memory coalescing and cache behaviour.

We present our efforts to establish a link between the performance differences shown in this chapter and graph structure in Chapters 5 to 7 on the next page, on page 85, and on page 99.

CHAPTER 5

Graph Generation

One of our starting assumptions is that the structure of graphs has a significant impact on performance. In the previous chapter we established that the run time of implementations can differ by an order of magnitude or more on the same graph. We also established that the best performing implementation differs from graph to graph. This leaves us to show that these performance differences can be attributed to the structure of these graphs.

This immediately leads us to the following problem: How do we define the "structure" of a graph? There are many properties describing the structure of graphs, the KONECT handbook [50], for example, identifies over 40 properties. However, there is no consensus on a standard set of properties for classifying graphs.

A complicating factor is that most of these structural properties are strongly correlated. Consider the case of the number of edges and the average degree of a graph. If we increase the number of edges, this clearly also increases the average degree of the graph. To the best of our knowledge there is no known set of independent properties that describes the structure/topology of a graph.

This lack of consensus leaves us to come up with our own set of properties to investigate. Our focus is on properties related to the degree distribution of the graph for two reasons. First, of all the possible properties

This chapter is based on work previously presented in:
Merijn Verstraaten et al. "Synthetic Graph Generation for Systematic Exploration of Graph Structural Properties". In: *"Euro-Par 2016: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2016, pp. 557–570. ISBN: "978-3-319-58943-5"

they are some of the easiest and fastest to compute. Second, there are several ways the degree of vertices can effect the efficiency of various parallelisation strategies, as discussed in Sections 2.3 and 4.2 on page 14 and on page 43.

In an ideal world we would have a comprehensive dataset of graphs, where we vary one of our structural properties at a time, keeping the others the same. For example, changing the average degree of vertices, without affecting the diameter, number of edges, etc. In that scenario, investigating which properties impact performance and by how much is straightforward. Simply run all implementations against each graph and relate changes to properties to change in performance.

However, it is unlikely that such a dataset exists — or even can exist — due to most structural properties being correlated. In the absence of such an ideal dataset, we need to work with what *is* available. Which means relying on either publicly available real world datasets or on synthetically generated graphs.

The SNAP [56] and KONECT [51] repositories used in the previous chapter, and rest of this thesis, are the two largest and most well-known public graph repositories. Other notable repositories include the Network Repository [78] and House of Graphs [12].

These repositories consist of a large variety of real world graphs, including various networks, such as road, communication, social, and collaboration networks, but also rating and metadata databases. These repositories include graphs ranging from 10 vertices and 14 edges to 68 million vertices and 2.6 billion edges.

As such, these repositories provide good coverage of a wide variety of different graphs. However, they suffer from a problem that is common in many real world datasets. The data is noisy. When comparing two graphs in these datasets, we see that nearly every structural property differs. This makes systematic investigation on the impact of specific properties infeasible.

The other method of getting input graphs for experiments is to use synthetic graph generators. The advantage of synthetic generators is that we can generate as many graphs as needed of any size that is needed. This lets us generate datasets that are less noisy than real world datasets; allowing a more systematic exploration of the impact of changes between graphs.

However, in practice, most generators can only generate graphs from a single class or a small set of classes. Thus, synthetic generators only help us do systematic comparison between graphs of a single "class". The definition of a "class of graphs" here is rather broad and informal. We will cover the generator specific definitions of "class" in more details in the related work, Section 5.1 on the facing page.

We can combine graphs from different generators to get a more diverse set of graphs, but this is not a panacea. Different generators can have overlap in the types of graphs they produce and parts of the potential graph search space may not be covered at all.

Earlier we observed that our main interest was in generating graphs with specific degree distributions, allowing a systematic investigation of how this impacts performance. Furthermore, we wanted to explore this impact in combination with other structural properties.

We set out to create a generator, as none of the existing graph generators serve our use case well. In this chapter we present the design and implementation of our generator [89].

The results of our generator were promising while generating small graphs, but degraded quickly as we scaled up to larger graphs. This meant we were unable to generate the dataset we would need for the kind of systematic performance benchmarking we had in mind.

We identified a problem with our initial implementation that can explain the degradation as we scale up the graph size, but addressing this issue requires a considerable amount of engineering time. This engineering investment and the uncertainty of success led us to abandon the graph generation approach in favour of using our data from the SNAP and KONECT datasets.

5.1 Related Work

Graph generation has been an active area of research since the '60s [30] and remains so to this day [1, 35]. This is the result of two complementary reasons:

1. Real data might be unavailable, because it is proprietary or cannot be obtained [59], or

2. a better understanding of what the essential features of a given type of graph are is wanted or needed. [19]

5.1.1 Analytical Models

One of the first and best known models for graph generation is the Erdös-Rényi model [30]. However, this model did not adequately capture the properties of many real world graphs. For example, it does not follow the power law degree distribution found in many real world graphs [2, 8, 77].

The Erdös-Rényi model was later subsumed as a special case of the R-MAT model [20], which can model power law degree distributions as well as deviations from it.

Kronecker graphs [57] use the Kronecker product of a matrix to generate graphs. The authors presented a tool for fitting the graphs generated using this method to the parameters of existing graphs, showing that the method can approximate real world graphs. Moreover, this model lets one use the fitted parameters to study the properties of graphs similar to the fitted input.

Neither the R-MAT nor the Kronecker approach gave us sufficient control over the degree distributions of the generated graphs. The Kronecker graph tools have an associated fitting tool, which can fit the degree distribution of a generated graph to match another graph. However, if we had all the necessary graphs available for fitting, we would not need to generate them in the first place.

5.1.2 Evolutionary Approaches

Generating graphs is also a topic of interest in the neural network community in AI. As the problem complexity in the field increased, so did the complexity of the used neural network topologies. Researchers started to look for methods to automate the generation of these complex networks. One of the more promising approaches, introduced in 1996, was NeuroEvolution of Augmenting Topologies (NEAT) [83, 84].

NEAT is based on evolutionary computing. The algorithm starts with a minimal neural network and incrementally adds vertices and edges to it. This works well, but encountered significant scaling issues.

This constructive approach of adding individual vertices and edges makes this approach time-consuming and memory intensive to produce large neural networks. Additionally, the larger the neural networks becomes, the harder it is to produce the complex topologies required to solve the AI problems encountered in the field.

Follow-up research tried to scale NEAT to neural networks of millions of vertices since the principle of NEAT works for small neural networks. The result of this research is a new algorithm, called HyperNEAT.

In HyperNEAT [36, 82] the evolutionary algorithm does not directly evolve the resulting neural network. Instead, it evolves a generating function for the eventual neural network. These generating functions are designed to more easily produce complex and recurring patterns, such as symmetry, anti-symmetry, and repetition with variation. In practice, HyperNEAT succeeds in producing neural networks that are orders of magnitude larger than those generated with NEAT.

The work on NEAT and HyperNEAT are not the only attempts to use evolutionary algorithms for graph generation. The work of both Bach et al. and Bailey et al. uses evolutionary algorithms to evolve generators/models for the generation of graphs [4, 5]. These generators consist of sequences of

operations that insert or construct certain motifs or permute the existing graphs.

Unlike NEAT, which evolves a single graph, and HyperNEAT, which evolves a generating function for a single graph, both Bach et al. and Bailey et al. evolve a graph generator. Producing a graph generator instead of a graph has two advantages: Instead of generating single graphs, we can produce sets of graphs. It is also easier to extend them to produce more complex patterns and structures.

Both papers show that they can successfully produce very diverse kinds of graphs. The produced generators use iterative, constructive approaches for generating graphs. A downside of this approach is that it is slow and memory intensive to generate larger graphs. The largest graphs generated in [5] are about 1,000 vertices; the generator from [4] already struggles with graphs of this size.

5.1.3 Comparative Analysis

The graph processing community's interest in understanding how algorithms behave on real graphs of different shapes and sizes is one of the main drivers of graph generation research. As a result, most of the field is focussed on analytical models for real world graphs, as described in Section 5.1.1 on page 69.

These models let researchers generate additional "real" input data for experiments, as well as help inform algorithm implementation methods. Unfortunately, this focus on mimicking specific classes of real world graphs means there has not been a lot of attention to more general and flexible generators.

Our interest in the impact of structural properties on algorithm performance is more abstract. The limited flexibility and reduced number of tunable parameters in the analytical model hinders our ability to use existing tools to generate the graphs we need.

Investigating the link between structural properties and performance does not require graphs that resemble real world data. Instead, we need datasets with less noise and variation between properties. Graphs in such datasets are unlikely to resemble real world graphs, but would simplify our investigation significantly.

The evolutionary approaches to graph generation discussed in Section 5.1.2 on the facing page are less focussed on mimicking specific types of real world graphs and show they can generate widely varying types of graphs. However, there are concerns regarding the scalability and controllability of these approaches.

To analyse the performance impact of structural properties we want to investigate the impact of these properties over a wide range of different graph sizes. Furthermore, we need to ensure that the compute times for

our algorithms are long enough that any performance differences between algorithm implementations do not get lost in the measurement noise. As a result, the inability for NEAT and the methods proposed by [4] and [5] to scale to larger graphs is worrisome.

Additionally, it is unclear whether HyperNEAT's generating functions or the motif- and generator-based approaches in [4] and [5] are sufficiently expressive to generate all the graphs of interest.

Finally, one of the steps in HyperNEAT consists of mapping a hypercube pattern (produced by the generating function) to a lower-dimensional space to obtain the actual graph. This is one the techniques that lets HyperNEAT generate such complex topologies. However, depending on the mapping method chosen, this can result in superlinear complexity. For scalability reasons it is preferable for the complexity to be linear in the number of vertices and/or edges.

5.2 A New Graph Generator Design

Neither existing real world datasets nor existing graph generators provide us with the kind of test data needed for the kind of sensitivity analysis we envision. In this section we discuss our requirements for a graph generator and how our algorithm design accommodates these requirements.

5.2.1 Requirements

Our end goal is to generate graphs that allow us to explore the impact of different graph properties systematically. Ideally, our generator is capable of generating graphs matching a user-specified set of desired values for structural properties — e.g., number of vertices, edges, connected components, and/or degree distribution. A generator well-suited for this type of systematic investigation should support:

- Fine granularity: vary as little as a single property at a time.
- Possibility to expand: add new structural properties.
- Scalability: generate small *and* large graphs within a reasonable time budget.

In essence, our graph generation problem translates to the search problem of finding graph(s) conforming to a set of structural properties in the search space of potential graphs. This search is complicated by the fact that many structural properties are either correlated or interdependent.

Evolutionary computing is well-known for its ability to efficiently search large spaces with complex interdependencies and/or correlations between parameters. And we are not the first to have this idea. Several of the

related research covered in Section 5.1 on page 69 use evolutionary computing for graph generation. However, different goals led them to make different design decisions in their application of these algorithms.

5.2.2 Evolutionary Computing for Graph Generation

Evolutionary computing is the collective name given to a range of techniques based on principles of natural evolution, such as natural selection and inheritance.

A key feature of evolutionary computing techniques is their ability to produce good results when dealing with large search spaces and large numbers of interdependent parameters; these properties makes evolutionary computing an appealing starting point for our problem.

The basic principle behind most evolutionary computing algorithms is simple:

1. Generate an initial population of candidate solutions.
2. Select a number of solutions for reproduction based on their quality.
3. Perform crossover[1] between selected solutions.
4. With a small probability, randomly mutate the candidate solutions.
5. Select survivors for the next generation based on quality.

There are endless variations on how to select parents, the probability of random mutations, how to select survivors, and how many new solutions should be generated in every generation. There are several standard choices that appear to work well for most algorithms, avoiding the need to perform substantial benchmarking to determine the right choices.

However, the remaining choices *are* problem specific and have a large impact on performance and success. Since evolutionary algorithms are stochastic, an important point of concern is the time it takes to converge to a set of acceptable result graphs.

As we want both fine-grained tuning and large scale graphs, we are faced with a large search space and long-running algorithm. For example, the larger graphs in SNAP have over 4 million vertices and 68 million edges (e.g., soc-Livejournal). Assuming a vertex-centric approach the search space for generating an undirected graph of 4 million vertices has a search space of $2^{4,000,000}$ possible unique graphs.

[1] With crossover parts of the "genetic code" of different solutions are recombined to form new solutions

Enumerating this search space is infeasible, even if we had an efficient heuristic for pruning uninteresting candidates. To obtain acceptable convergence speeds we have to ensure that the primitives used for generating new candidates cover enough of the search space quickly.

The key idea behind the crossover operation is that it combines successful or interesting parts of solutions, resulting in an even better solution. As such, selecting a good crossover operation is critical to acceptable convergence speeds.

Another important choice is the rate of mutation: too low and the algorithm takes too long to explore promising related solutions; too high and the algorithm may never converge on any optimal points, continuously hopping over them.

5.2.3 Candidate Graph Representations

How to represent candidate solution graphs is an essential choice for graph generation, because it impacts which crossover and mutation primitives we can efficiently implement. Several sensible choices exist:

- Individual graphs represented as a connectivity matrix.
- Individual graphs represented as an edge list.
- Generating functions, i.e., a function that generates one specific graph.
- Graph generators, i.e., a generator that generates graphs according to some patterns.

5.2.3.1 Connectivity Matrices

A connectivity matrix represents a graph by encoding it as a 2-dimensional matrix. A graph G of N vertices is represented as an $N \times N$ matrix. An edge (n, m) in G is encoded as $G_{n,m} = 1$. All matrix elements without a corresponding edge are 0.

In this representation the most straightforward implementation of mutation consist of randomly inserting or deleting edges. This is implemented by generating random indices and flipping them from 0 to 1 or vice versa.

For crossover there are three simple methods:

Edge-wise, for every index in the connectivity matrix, randomly select a parent and keep its value.

Vertex-wise, for every vertex in the graph, randomly select a parent and keep the edges associated with that vertex.

Single-point, select a random point in the matrix and for every edge before it, keep the edges of the first graph, for the remaining edges, keep those of the second graph.

Edge-wise crossover results in a very thorough mixing of two candidate solutions, keeping roughly 50% of the edges from the first and 50% of the second parent. Additionally, it is easy to implement. However, indiscriminately picking edges from either parent will almost certainly destroy any interesting subsections of the graph, the exact thing that crossover is supposed to maintain.

Vertex-wise crossover preserves significantly more structure from the individual parents, since all vertices from one parent will keep parts of their environment from that parent.

Single-point crossover is even more conservative, since it always preserves sequential sets of vertices. However, since vertices are not necessarily sequentially connected it is unclear if this preserves significantly more structure than vertex-wise crossover would. It is also unclear whether this actually produces a net benefit for convergence.

With this representation it is easiest to keep the number of vertices constant. Mutations that insert or delete vertices are not particularly hard to implement, but these change the size of the matrix representing the graph. This, in turn, complicates the crossover primitive, as it is unclear how the above crossover primitives should be interpreted when performing crossover on two matrices of differing sizes.

5.2.3.2 Edge Lists

As the name implies, an edge list represents a graph as a list of edges — i.e., a sequence of vertex pairs. Edge insertion or deletion mutations are as easy to implement as they are for connectivity matrices; simply delete a pair from or insert a pair into the sequence.

Additionally, edge lists admit a new mutation primitive that is hard to implement for connectivity matrices: remapping edges. We can pick a random edge in the sequence in change either its origin or destination to another random vertex. We can even extend this mutation to allow it to remove or create new vertices.

The crossover primitives for connectivity matrices apply equally well for edge lists. Edge-wise crossover iterates over both parents sequences and picks a random one for the new graph. Vertex-wise crossover requires slightly more implementation work, but remains conceptually the same. Single-point crossover is straightforward to implement too: We pick a random index in the sequence, for parent 1 keep all pairs before that index and for parent 2 we keep all pairs after that index.

The biggest difference with connectivity matrices is that certain restrictions are easier to impose on the generation process. For example, with edge lists it is simpler to generate graphs with a fixed number of edges. Start with the desired number of edges and do not use any mutations that insert or delete edges. That is, only allow mutations that change the origin and/or destination of existing edges.

On the other hand, edge list representations make it more complex to guarantee that the number of vertices in a graph stays constant across crossover and/or mutation. This property is simple to maintain when performing crossover and/or mutation on connectivity matrices. Thus, the two representations are complementary depending on the kind of graphs we wish to generate.

5.2.3.3 Generating Functions

The biggest problem with the straightforward representations proposed above is that their entirely random permutations can take a long time to converge on more complicated structures that might be needed to achieve the desired values for more properties, such as clustering coefficient.

One solution is to not evolve a graph directly, but rather evolve a function that generates a graph. These generating functions are made by combining smaller, independent functions together. The idea is that these smaller functions capture part of the graph's structure. This makes it easier to generate more complex structures and recombine them into new graphs, hopefully speeding up the convergence to desired result graphs.

HyperNEAT, for example, uses this technique. HyperNEAT evolves Compositional Pattern Producing Networks (CPPNs); these are, essentially, generating functions for neural networks. These CPPNs were designed to be effective at generating complex structures, such as symmetries, repetition, and repetition with variation. The results from HyperNEAT show that this method can successfully produce complex neural networks. However, there are two important concerns about this representation.

First, the indirection of first running the generating function to produce a graph, means that the performance of this generating function becomes a crucial bottleneck in the algorithms speed. There are no details on HyperNEAT's performance in terms of wall-clock time, which makes it unclear how effective it would for generating large numbers of graphs.

Second, while CPPNs have been shown to be capable of generating complex graphs, this does not mean they are flexible enough to generate any possible graph. It is an open question whether CPPNs are general enough for the systematic benchmarking we would like to do.

5.2.3.4 Generators

All the previous representations deal with a single graph, but this is not the only approach we can take. During our discussion of generating functions we implicitly assumed these functions where deterministic. If we built functions out of stochastic components instead, we do not get a generating function for a single graph, but instead one that can generate many similar graphs.

This approach was proven possible by Bailey et al. [5] and Bach et al. [4]. However, their approach raises the same concerns as generating functions: It is unclear whether the evolved generators are sufficiently flexible and expressive to generate all graphs of interest.

Additionally, the generators have to be fast enough to be able to quickly evaluate their suitability. Initial tests showed that implementation used by Bach et al. has scalability problems, even for very small graphs ($\sim$1,000 vertices). Similarly, in [5] there is no discussion about which types of graphs are ruled out by the design of their generator. Additionally, the authors do not seem to have tried to scale the approach beyond a few 100 vertices.

5.3 Implementation

Our prototype implementation evolves connectivity matrices directly. The reasons for using connectivity matrices over edge lists are twofold:

1. It was simpler to implement with our existing file format, and
2. we are more interested in generating graphs with a fixed and predictable number of vertices.

The first step with any evolutionary algorithm is to define the fitness function. The purpose of this function is to map candidate solutions to a quality metric that indicates the quality of the candidate solution. We are mostly interested in generated graph with specific degree distributions, so our focus for the generator is to produce weakly connected graphs with specific degree distributions.

Matching the distribution of degrees in a graph to a specific distribution is straightforward. The Kolmogorov-Smirnov (KS) test for goodness of fit [64] compares an Empirical Distribution Function (EDF) with a Cumulative Distribution Function (CDF) and gives us the absolute difference between the two.

Our null hypothesis is that the EDF and CDF come from the same distribution. This null hypothesis should be rejected if the absolute difference between the EDF and CDF exceeds our critical value. The critical

Significance	Critical Value
$p = 0.20$	$1.07/\sqrt{N}$
$p = 0.15$	$1.14/\sqrt{N}$
$p = 0.10$	$1.22/\sqrt{N}$
$p = 0.05$	$1.36/\sqrt{N}$
$p = 0.01$	$1.63/\sqrt{N}$

Table 5.1: Critical values for KS goodness-of-fit test, with sample size $N > 35$.

value depends on both the desired significance and number of samples. In Table 5.1 we list appropriate critical values for sample sizes larger than 35.

We also need a fitness metric for the weak connectivity of graphs. A simple and efficiently computable metric is the percentage of the graph's vertices that are in the same weakly connected component[2].

The connectivity and KS metrics above give us two independent fitness metrics, which we need to reduce to a single metric that can be easily compared for our evolutionary algorithm. Two common ways to combine multiple metrics into one are the weighted sum of metrics or using a Pareto ranking.

Initial experiments showed that the weak connectivity of *all* generated graphs converges to 1 in a few generations. At which point the KS goodness of fit becomes the sole method of ranking graphs. So, we opt for a simpler solution: We treat the connectivity percentage as value between 0 and 1 and multiply the KS goodness of fit with this value. This way the weak connectivity functions as a penalty whenever it drops below 1.

Our final algorithm design consists of:

Population size: 100 candidate solutions.

Parent selection: Weighted random selection.

Number of children: 100 new children every generation.

Crossover: Edge-wise, vertex-wise, and single-point.

Mutation rate: $\frac{1}{0.1N^2}$, where N is the number of vertices.

Survivors: Keep best 20% plus weighted random selection.

Our initial population consists of 100 randomly generated graphs. We create these initial graphs by computing the approximate number of edges expected for a given number of nodes and a distribution, and then uniform randomly setting that many edges in our connectivity matrix.

[2] Specifically, the weakly connected component containing vertex 0

To generate 100 new children we use weighted random selection to pick 100 pairs of graphs from the current population to serve as parents. We perform crossover between these two parents, followed by uniform random mutation of the edges in the resulting graph.

Finally, we merge the child and parent populations. The survivor population is created by taking the top 20% of the combined population, and supplementing it with a weighted random selection of the remaining 80%.

Our choice of population size, parent selection mechanism, and number of children are standard values in the evolutionary computing literature. These values provide decent results for most problems and adjusting them is unlikely to improve convergence speed or result quality much. Our choices for crossover, mutation, and survivor selection *are* important for convergence and result quality.

Mutations are responsible for exploring the neighbourhood of existing candidate solutions. With a high mutation rate candidate solutions drift apart at a higher rate, diverging from their parents at a faster rate, speeding up the exploration of the search space. A low mutation rate explores the search space around existing solutions more thoroughly.

If the mutation rate that is too low turns the algorithm into an inefficient exhaustive search; a mutation rate that is too high results in instability and risks skipping over good solutions.

Crossover combines, hopefully successful, sections of different graphs together, exploring the search space between those two graphs. The idea is that some children will combine valuable, differing parts of both parents into a single graph, producing a graph with a better fitness than either parent.

However, this does require crossover to preserve enough "valuable" substructure. We implemented each of the crossover methods discussed in Section 5.2 on page 72 to investigate which approach is more effective.

5.4 Results

There are two main criteria for evaluating our graph generator:

1. The success rate, i.e., its ability to find graphs matching our requirements, and

2. the convergence rate, i.e., the speed with which solutions are found.

The success and convergence rates are closely related. A generator that produces the desired graphs, but takes 1,000 years to do so is, for all practical purposes, indistinguishable from a generator that does not succeed at all…

For our first test we wanted to show that our generator can successfully produce different degree distributions. These tests were done with small

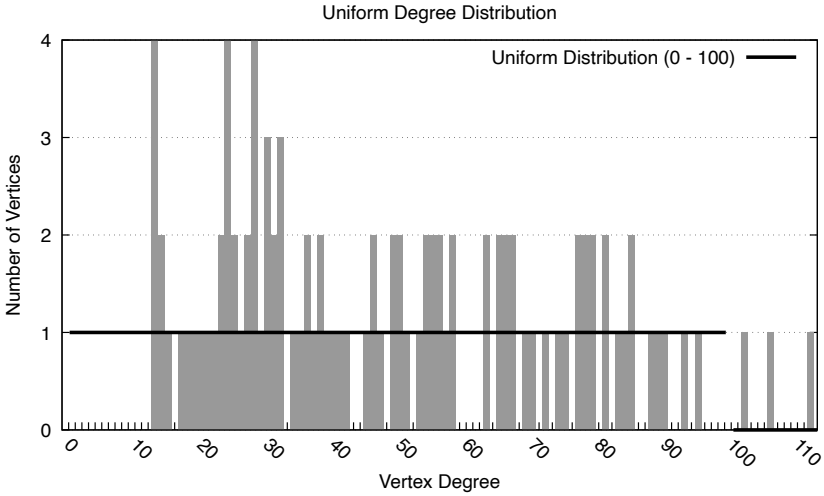


Figure 5.1: Empirical degree distribution of graph generated with uniform degree distribution (0–100).

graphs of 100 vertices to ensure fast convergence. In Figs. 5.1 to 5.3 on pages 80–81 we show the actual degrees and expected values for uniform, exponential, and Gaussian distributions respectively.

These results are after 1,000 generations, but for graphs this small comparable fitness values start occurring after 100–200 generations. The empirical distributions are a good fit for the target distributions, per our KS tests. Thus, we conclude that our generator is able to generate graphs whose degree distributions match our three target distributions. We do not see any theoretical reasons why this would not generalise to other distributions.

Having shown that our generator can produce graphs with different distributions, it is time to consider how long it takes to produce these result graphs. It should be self-evident that the time T to produce a (set of) graphs is:

$$T = N \times T_{\text{gen}}$$

Where N is the number of generations until a result and T_{gen} is the time it takes to handle one generation. That is, the time it takes to generate a set of children, evaluate their fitness, and select survivors from the children and parents.

Our chosen population size is only 100, so the survivor selection takes a negligible amount of time and can safely be ignored. The main computational cost for a generation is the creation of new children and the

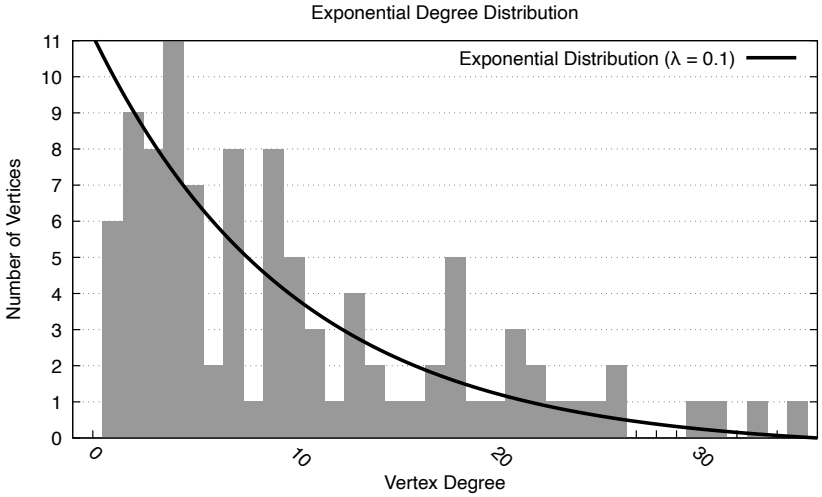


Figure 5.2: Empirical degree distribution of graph generated with exponential degree distribution ($\lambda = 0.1$).

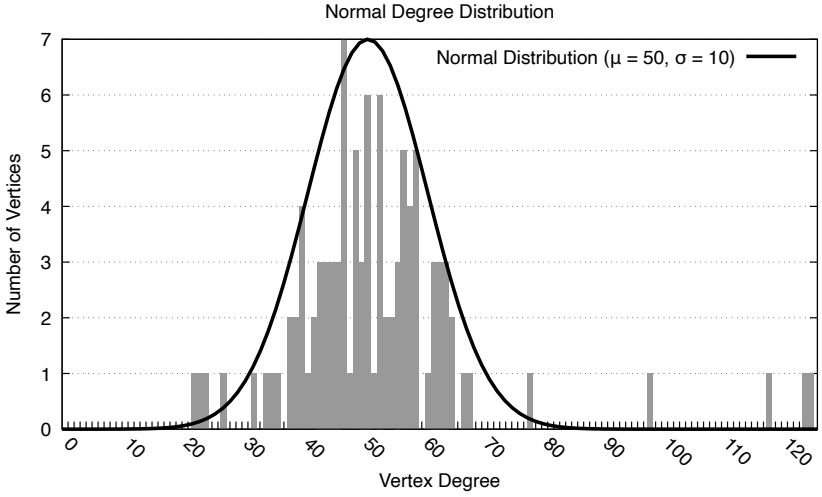


Figure 5.3: Empirical degree distribution of graph generated with normal degree distribution ($\mu = 50$, $\sigma = 10$).

computation of their fitness. Both of these operations are trivially parallelisable, which means that T_{gen} is proportional to the time it takes to create and evaluate one new graph.

The number of required generations N depends on how effective our primitives are at finding the promising parts of the search space. In other words, the convergence rate of our generator is determined by two factors: The scalability of our primitives and their effectiveness.

The end goal for our prototype is to generate graphs that are big enough to require a significant amount of compute time on the Graphical Processing Unit (GPU). If the compute time is too short, any performance differences due to structural properties are lost in the measurement noise.

It is unclear exactly how large our graphs need to be. The largest graphs in SNAP and KONECT consist of millions of vertices and billions of edges. The median seems to be on the order of hundreds of thousands of vertices and millions of edges, so we expect to need graphs of at least that size.

For a graph of 300,000 vertices, the entire process of crossover, mutation, and fitness computation takes 3 seconds. This time increases to approximately 35 seconds for a graph of 3,000,000 vertices. For 1,000 generations at these sizes, we get a total run time of 50 minutes at the small end and up to 10 hours at the large end. This means graphs of millions of vertices are on the edge between feasible and infeasible to generate.

The above time frames assume that we find acceptable graphs in 1,000 or fewer generations at these scales. The 100 vertex graphs in Figs. 5.1 to 5.3 on pages 80–81 converged in 100–200 generations. For graphs of 1,000 vertices we see rapid improvements in fitness for the first 100 generations, after which the improvements plateau for a long time. It takes well over 1,000 generations before the correspondence between graphs and target distribution becomes statistically significant.

For even bigger graphs we do not reach statistical significance at all. Again, we see rapid improvement in fitness for the first ∼200 generations, after which we hit a plateau. Improvement over this plateau is slow enough that the algorithm does not terminate. At least, not in the time frame that we allowed the algorithm to run.

This raises an important question: Are these plateaus a result of the fundamental difficulty of producing graphs that meet our requirements *or* are our crossover and mutation primitives insufficient?

The importance of good crossover and mutation primitives is well-known in the evolutionary computing world. This was reconfirmed by our own experiments with different combinations of crossover and mutation rates. In our experiments we saw that edge-wise crossover takes considerably longer to converge to good results, even on small graphs. With vertex-wise crossover the required number of generations was 1–2 orders of magnitude lower.

The biggest concern with our current primitives is our implementation of mutation. Our implementation is a simplistic uniform random mutation on all the potential edges. A graph with N vertices has an $N \times N$ connectivity matrix of potential edges. We mutate the connectivity matrix by selecting a uniform random number of these potential edges and flipping them. Changing non-existent edges into edges and vice versa.

The unforeseen flaw of this approach is that it implicitly biases our graph generation towards graphs where 50% of the connectivity matrix is made up of edges. The ratio of edges and non-edges in our uniform random sample reflects the ratio in the overall connectivity matrix. Combined with our "bit flip" mutation mechanics, it follows that a sample that has less than 50% edges will introduce more edges than it removes. Vice versa, a sample with *more* than 50% edges will remove more than it introduces. Over time this converges to the equilibrium state where 50% of the connectivity matrix consists of edges. At which point we expect all samples to have 50% edges and have no net effect on the number of edges.

5.5 Conclusion

We set out to build a synthetic graph generator to generate *graphs with specific structural properties*, as the existing analytical models and graph generators, discussed in Section 5.1 on page 69, did not accommodate the structural properties we were interested in. Our goal was to use these graphs to perform a systematic exploration of the impact of structural properties on performance.

Most structural properties of graphs are correlated and it is not always clear whether a graph with a predefined set of properties exists. Our proposed generator is based on evolutionary computing, a search method that is known to produce good results in the presence of complex, correlated, or even conflicting requirements. Additionally, previous work has shown promise for evolutionary computing in graph generation (see Section 5.1.2 on page 70).

In Section 5.4 on page 79 we show that our proposed generator is capable of generating graphs with specific degree distributions at smaller graph sizes, but runs into trouble as we scale up to larger graph sizes. We show that the primitives are fast enough for graphs up to 300 thousand or 3 million vertices *iff* the generator converges to an acceptable solution in a reasonable number — e.g., around 1,000 — of generations.

At these sizes, 1,000 generations take about 50 minutes to 10 hours to run with maximal parallelism. However, as we scaled up our experiments to larger graph sizes (10,000–100,000 vertices) we ran into problems. After 100 generations of rapid improvement, the graph quality plateaus and does not improve to an acceptable quality in the remaining 900 generations.

Even doubling our experiments to 2,000 generations, doubling our run time too, we were unable to get acceptable results. With a run time of 20 hours or more — assuming maximal parallelism — it was infeasible for us to use this method of generation for our GPU experiments.

The likely root cause of this plateauing effect is that the crossover and mutation primitives are not sufficient. In Section 5.4 on page 79 we noted that uniform random mutation tends to a graph where 50% of potential edges are present; this makes very sparse and very dense graphs unlikely to occur and rules out large portions of the search space.

One way to address the bias in our mutation primitive is to switch from a "bit flip" mutation to a primitive where the odds of inserting an edge and the odds of removing an edge are independent. Another interesting variation would be to include these mutation rates in the representation. This would allow the mutation rate to evolve along the graphs and vary as part of the search. Similarly, we might want to experiment with crossover implementations that preserve more structure than the existing primitives.

There are practically infinitely many avenues to tweak the crossover and mutation approaches. These would, hopefully, allow us to successfully produce the larger size graphs we need for our experiments. This exploration makes for interesting research, but takes a considerable amount of engineering.

The required engineering effort, combined with the uncertainty of any results led us to abandon the graph generation avenue in favour of our existing KONECT and SNAP datasets in Chapters 4, 7, and 8 on page 41, on page 99, and on page 111.

CHAPTER 6

Analytical Performance Modelling

In Chapter 4 on page 41 we highlight how the performance of implementations varies dramatically across input graphs. We use analytical modelling to try to understand the relationship between input graph and the observed performance.

In this chapter we present our PageRank workload model and our efforts to predict the parallel performance of our PageRank implementations from this workload. We focus on PageRank as it is more regular than Breadth-First Search (BFS) and therefore easier to model.

6.1 Workload Models

The computational workload of PageRank is negligible, as shown by the implementations in Section 4.4.1 on page 51. Like BFS, and many other graph algorithms, the PageRank kernels consist mostly of reading and writing memory. Because of this memory-bound behaviour, we focus on modelling the memory access behaviour.

We start by creating a workload model for each implementation. In this workload model we only consider the total operations performed, ignoring

This chapter is based on work previously presented in:
Merijn Verstraaten et al. "Quantifying the Performance Impact of Graph Structure on Neighbour Iteration Strategies for PageRank". In: *"Euro-Par 2015: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2015, pp. 528–540. ISBN: "978-3-319-27308-2"

Graphical Processing Unit (GPU) parallelism for now. We identify three classes of memory accesses: global memory reads, global memory writes, and global atomic operations.

Within these three classes there is still a lot of variability. Two accesses within the same class can have wildly different access times due to cache effects and/or atomic contention. For now, we ignore these actual performance cost of operations and treat them as unknown constants.

We represent the cost of a random global read as T_{read}, the cost of a random global write as T_{write}, and the cost of a global atomic add operation as T_{atom}.

6.1.1 Abstract Workload Models

For our abstract workload models we refer back to pseudocode in Section 4.4.1 on page 51. Our edge list kernel in Algorithm 1 on page 51 uses one thread per edge, performing 4 reads: 2 to find the destination and origin, 1 to get the origin's degree, and 1 to get the old PageRank value. Finally, each thread performs an atomic addition to update the new PageRank value of the destination vertex, resulting in:

$$\begin{aligned} T_{\text{edge}} &= \sum_{e \in E} (4 * T_{\text{read}} + T_{\text{atom}}) \\ &= 4 * |E| * T_{\text{read}} + |E| * T_{\text{atom}} \end{aligned}$$

With vertex push we use one thread per vertex. In Algorithm 2 on page 52 we see that each thread performs 2 reads: 1 to read the vertex' degree and 1 to read its PageRank value. This is followed by d atomic addition operations, where d is the degree of that vertex. The number of operations performed by vertex push thus boil down to:

$$\begin{aligned} T_{\text{push}} &= \sum_{v \in V} (2 * T_{\text{read}} + d_v * T_{\text{atom}}) \\ &= 2 * |V| * T_{\text{read}} + |E| * T_{\text{atom}} \end{aligned}$$

In Algorithm 3 on page 52 we see that the vertex pull kernel performs 2 reads for each of its vertex' neighbours, and then performs a non-atomic write to store the new PageRank result. The total operations performed by vertex pull thus boil down to:

$$\begin{aligned} T_{\text{pull}} &= \sum_{v \in V} (2 * d_v * T_{\text{read}} + T_{\text{write}}) \\ &= 2 * |E| * T_{\text{read}} + |V| * T_{\text{write}} \end{aligned}$$

The consolidation kernel, shown in Algorithm 6 on page 54, is the same for each of the above kernels, performing 2 reads: one for the new incoming rank value and one for the old PageRank value, followed by an atomic addition and 2 writes to store the new PageRank and reset the incoming rank:

$$\begin{aligned} T_{\text{con}} &= \sum_{v \in V} (2 * T_{\text{read}} + 2 * T_{\text{write}} + T_{\text{atom}}) \\ &= 2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}} \end{aligned}$$

For our NoDiv implementation of vertex pull, shown in Algorithm 7 on page 55, the workload model is:

$$\begin{aligned} T_{\text{NoDiv}} &= \sum_{v \in V} (d_v * T_{\text{read}} + T_{\text{write}}) \\ &= |E| * T_{\text{read}} + |V| * T_{\text{write}} \end{aligned}$$

The corresponding consolidation, shown in Algorithm 9 on page 56 boils down to:

$$\begin{aligned} T_{\text{conNoDiv}} &= \sum_{v \in V} (3 * T_{\text{read}} + 2 * T_{\text{write}} + T_{\text{atom}}) \\ &= 3 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}} \end{aligned}$$

We can now combine the workload of the update and their correspond-

ing consolidation kernels and get:

$$\begin{aligned} T_{\text{edge}} &= (4 * |E| * T_{\text{read}} + |E| * T_{\text{atom}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}}) \\ &= (4 * |E| + 2 * |V|) * T_{\text{read}} + 2 * |V| * T_{\text{write}} \\ &\quad + (|V| + |E|) * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{push}} &= (2 * |V| * T_{\text{read}} + |E| * T_{\text{atom}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}}) \\ &= 4 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + (|V| + |E|) * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{pull}} &= (2 * |E| * T_{\text{read}} + |V| * T_{\text{write}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}}) \\ &= 2 * (|E| + |V|) * T_{\text{read}} + 3 * |V| * T_{\text{write}} + |V| * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{NoDiv}} &= (|E| * T_{\text{read}} + |V| * T_{\text{write}}) \\ &\quad + (3 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + |V| * T_{\text{atom}}) \\ &= (3 * |V| + |E|) * T_{\text{read}} + 3 * |V| * T_{\text{write}} + |V| * T_{\text{atom}} \end{aligned}$$

6.1.2 Concrete Workload Models

The abstract workload models above are based on the pseudocode descriptions of our implementations. The real world is messy and accommodating it can force our real implementations to deviate from the simplicity of our pseudocode. In this subsection we contrast the abstract workload models derived from the pseudocode above, with the actual memory accesses found in the Parallel Thread Execution (PTX) assembly generated by the Compute Unified Device Architecture (CUDA) compiler.

The simplicity of our PageRank kernels makes it straightforward to do this analysis manually. The relevant PTX assembly can be found in Appendix B on page 151. We can simply count load (`ld`) and store (`st`) instructions, accounting for repetitions due to loops is straightforward with the reference C++ included in the assembly.

Note that our kernels were written using "grid-stride loops" [40] to accommodate workloads larger than the maximum grid size. However, none of the graphs in our datasets exceed this size, so we can ignore them in this analysis.

6.1.2.1 Edge List

Appendix B.1 on page 151 shows the PTX assembly for our edge list implementation. In this assembly listing we identify the following operations: 1 load per edge of the `u64` edge count; 2 loads per edge of the `u64` edge array pointers; 2 loads per edge of the `u32` the origin and destination vertices; 1 load per edge of the `u32` degree of the origin vertex; 1 load per edge of the `f32` origin vertex PageRank value; and 1 atomic add per edge of the `f32` destination vertex PageRank value. Giving:

$$T_{\text{edge}} = 7 * |E| * T_{\text{read}} + |E| * T_{\text{atom}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{edge}} = {} & 40 \text{ byte} * |E| * T_{\text{read}} \\ & + 4 \text{ byte} * |E| * T_{\text{atom}} \end{aligned}$$

6.1.2.2 Vertex Push

The PTX assembly listing for vertex push, shown in Appendix B.2 on page 154, we find: 1 load per vertex of the `u64` vertex count; 2 loads per vertex of the `u64` vertex and edge array pointers; 2 loads per vertex of the `u32` vertex offsets in the edge array; 1 load per vertex of the `f32` PageRank value; 1 load per outgoing edge of the `u32` destination vertex; and 1 atomic add per outgoing edge of the `f32` destination PageRank value. Giving:

$$T_{\text{push}} = 6 * |V| * T_{\text{read}} + |E| * T_{\text{read}} + |E| * T_{\text{atom}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{push}} = {} & 36 \text{ byte} * |V| * T_{\text{read}} \\ & + 4 \text{ byte} * |E| * T_{\text{read}} \\ & + 4 \text{ byte} * |E| * T_{\text{atom}} \end{aligned}$$

6.1.2.3 Vertex Pull

Our vertex pull listing, shown in Appendix B.3 on page 162, we find: 1 load per vertex of the `u64` vertex count; 2 loads per vertex of the `u64` reverse vertex and edge array pointers; 2 loads per vertex of the `u32` vertex offsets in the edge array; 2 loads per incoming edge of the `u32` origin vertex and its degree; 1 load per incoming edge of the `f32` PageRank value; and 1 store per vertex of the `f32` new PageRank value. Giving:

$$T_{\text{pull}} = 5 * |V| * T_{\text{read}} + 3 * |E| * T_{\text{read}} + |V| * T_{\text{write}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{pull}} = {} & 32 \text{ byte} * |V| * T_{\text{read}} \\ & + 12 \text{ byte} * |E| * T_{\text{read}} \\ & + 4 \text{ byte} * |V| * T_{\text{write}} \end{aligned}$$

6.1.2.4 Vertex Pull NoDiv

The NoDiv listing of vertex pull, shown in Appendix B.4 on page 170, has: 1 load per vertex of the `u64` vertex count; 2 loads per vertex of the `u64` reverse vertex and edge array pointers; 2 loads per vertex of the `u32` vertex offsets in the edge array; 1 load per incoming edge of the `u32` origin vertex; 1 load per incoming edge of the `f32` PageRank value; and 1 store per vertex of the `f32` new PageRank value. Giving:

$$T_{\text{NoDiv}} = 5 * |V| * T_{\text{read}} + 2 * |E| * T_{\text{read}} + |V| * T_{\text{write}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{NoDiv}} = {} & 32 \text{ byte} * |V| * T_{\text{read}} \\ & + 8 \text{ byte} * |E| * T_{\text{read}} \\ & + 4 \text{ byte} * |V| * T_{\text{write}} \end{aligned}$$

6.1.2.5 Consolidate

Our consolidate implementation, shown in Appendix B.5 on page 177, performs: 2 loads per vertex of the `f32` old and new PageRank; 2 stores per vertex of the `f32` old and new PageRank; and 1 atomic add per 32 vertices. Giving:

$$T_{\text{con}} = 2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{con}} = 8\ \text{byte} * |V| * T_{\text{read}} \\ + 8\ \text{byte} * |V| * T_{\text{write}} \\ + 4\ \text{byte} * \frac{|V|}{32} * T_{\text{atom}} \end{aligned}$$

6.1.2.6 Consolidate NoDiv

And for our NoDiv version of consolidate, shown in Appendix B.5 on page 177, we find: 2 loads per vertex of the `f32` old and new PageRank; 2 stores per vertex of the `f32` old and new PageRank; 1 load per vertex of the `u32` vertex degree; and 1 atomic add per 32 vertices. Giving:

$$T_{\text{conNoDiv}} = 3 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}$$

or, expressed in bytes read or written, rather than individual reads or updates:

$$\begin{aligned} T_{\text{conNoDiv}} = 12\ \text{byte} * |V| * T_{\text{read}} \\ + 8\ \text{byte} * |V| * T_{\text{write}} \\ + 4\ \text{byte} * \frac{|V|}{32} * T_{\text{atom}} \end{aligned}$$

6.1.2.7 Concrete Totals

We can combine the concrete models for the update and consolidation kernels to get the total memory accesses in the PTX assembly, as shown below:

$$\begin{aligned} T_{\text{edge}} &= (7 * |E| * T_{\text{read}} + |E| * T_{\text{atom}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\ &= (7 * |E| + 2 * |V|) * T_{\text{read}} + 2 * |V| * T_{\text{write}} + (|E| + \frac{|V|}{32}) * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{push}} &= (6 * |V| * T_{\text{read}} + |E| * T_{\text{read}} + |E| * T_{\text{atom}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\ &= (8 * |V| + |E|) * T_{\text{read}} + 2 * |V| * T_{\text{write}} + (|E| + \frac{|V|}{32}) * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{pull}} &= (5 * |V| * T_{\text{read}} + 3 * |E| * T_{\text{read}} + |V| * T_{\text{write}}) \\ &\quad + (2 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\ &= (7 * |V| + 3 * |E|) * T_{\text{read}} + 3 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}} \end{aligned}$$

$$\begin{aligned} T_{\text{NoDiv}} &= (5 * |V| * T_{\text{read}} + 2 * |E| * T_{\text{read}} + |V| * T_{\text{write}}) \\ &\quad + (3 * |V| * T_{\text{read}} + 2 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}}) \\ &= (8 * |V| + 2 * |E|) * T_{\text{read}} + 3 * |V| * T_{\text{write}} + \frac{|V|}{32} * T_{\text{atom}} \end{aligned}$$

6.1.3 Relation Between Workload Models and Performance

Now that we have both our abstract workload models and their concrete counterparts, we can compare them and see what the deviation between pseudocode and reality is. Comparing the models from the previous two sections, we find the following differences between the abstract and concrete models:

$$\begin{aligned} T_{\text{edge}} &= 3 * |E| * T_{\text{read}} - \frac{31}{32} * |V| * T_{\text{atom}} \\ T_{\text{push}} &= (4 * |V| + |E|) * T_{\text{read}} - \frac{31}{32} * |V| * T_{\text{atom}} \\ T_{\text{pull}} &= (5 * |V| + |E|) * T_{\text{read}} - \frac{31}{32} * |V| * T_{\text{atom}} \\ T_{\text{NoDiv}} &= (5 * |V| + |E|) * T_{\text{read}} - \frac{31}{32} * |V| * T_{\text{atom}} \end{aligned}$$

We can eliminate the common factors from these equations, as these cannot affect the ranking of implementations compared to the abstract models. After eliminating these common factors we are left with the following deviation between abstract and concrete models:

$$\begin{aligned} T_{\text{edge}} &= 2 * |E| * T_{\text{read}} \\ T_{\text{push}} &= 4 * |V| * T_{\text{read}} \\ T_{\text{pull}} &= 5 * |V| * T_{\text{read}} \\ T_{\text{NoDiv}} &= 5 * |V| * T_{\text{read}} \end{aligned}$$

In general, we expect $|E|$ of a graph to be significantly larger than $|V|$. If we look at both the abstract and concrete workload models from

the previous sections with this in mind, then we see that the edge-centric implementation perform many more operations than all the other implementations. Looking at the discrepancies between the abstract and concrete workload models, we see that these only exacerbate this difference in workload.

However, when we look at the performance data for PageRank in Section 4.4.2 on page 54 we see that the edge-centric implementation is one of the best performing implementations overall.

This confirms that the parallelisation strategy has considerable impact on the overall performance of our GPU graph algorithms[1]. In order to conclude anything about performance from our workload models we need to model how each implementation parallelises its workload on the GPU.

6.2 Parallelising Workload Models

The most naive approach to modelling the parallelisation of our workload models is: take the total workload and divide it by the number of parallel execution units. However, this approach clearly does not work. The resulting ranking would be identical to the total workload ranking, as we are using the same hardware and same number of parallel execution units for each implementation.

Clearly, our workload parallelisation needs to account for implementation specific run time behaviour. The difficulty in approximating this run time behaviour mainly stems from two factors:

1. Workload imbalance between threads within a warp, and
2. non-uniform memory access times due to coalescing, caching, and atomic contention.

In Section 2.3 on page 14 we already covered how vertex-centric parallelisation strategies are susceptible to efficiency loss due to load imbalance. This happens when push or pull kernels process vertices with differing degrees within a single warp. The loop processing the outgoing/incoming edges diverges, causing some cores in the warp to be idle. On the other hand, the edge-centric version does not suffer from divergence at all, at the cost of performing more memory accesses, as shown by our workload models.

In Section 6.1 on page 85 we pretended the cost of reading or writing a memory location is constant. In reality, the memory subsystem on modern GPUs performs all kinds of tricks that make this untrue.

Threads within a single warp accessing the same or adjacent memory can have their accesses coalesced into a single global memory operation. Or

[1] It is always reassuring when obvious things are true…

Id	Ordering
1	Unordered
2	Absolute degree
3	Absolute degree (Pessimal)
4	In degree
5	In degree (Pessimal)
6	Out degree
7	Out degree (Pessimal)

Table 6.1: Memory orderings of `actor-collaboration` used in Fig. 6.1 on the next page. Pessimal orderings arrange the vertices for maximal difference in degree within each GPU warp.

the global memory operation can be avoided altogether, if that memory is already loaded into one of the caches. The result of this is that the access pattern of all the threads has a non-linear impact on the cost of reads and writes.

This can help explain the good performance of the edge list implementation in Section 4.4.2 on page 54. The edge list implementations are very likely to do coalesced access as a result of the memory layout of the data structure used by them.

For atomic operations the cost is dependent on the amount of contention. Concurrent atomic operations need to be serialised to obtain correct behaviour, which leads to atomic operations becoming slower when multiple threads try to atomically update the same memory location.

6.2.1 Static Approximation of Performance

From the above it should be clear that we cannot have accurately cost of T_{read}, T_{write}, and T_{atom} without modelling the entire dynamic scheduling and caching behaviour of the GPU running the code. However, to predict the best performing implementation we only require an approximation that is "accurate enough" to correctly rank our workloads.

To do this we need to approximate the cost of T_{read}, T_{write}, and T_{atom} per implementation.

These approximations need to be per implementation; following the discussion in Section 6.1.3 on page 92. There we observed that the workload model for edge list always does the most work, despite being one of the fastest overall implementation.

This leaves the question: Are the access patterns of each implementation static enough that we can, for a given GPU, approximate these costs for each of our workload models.

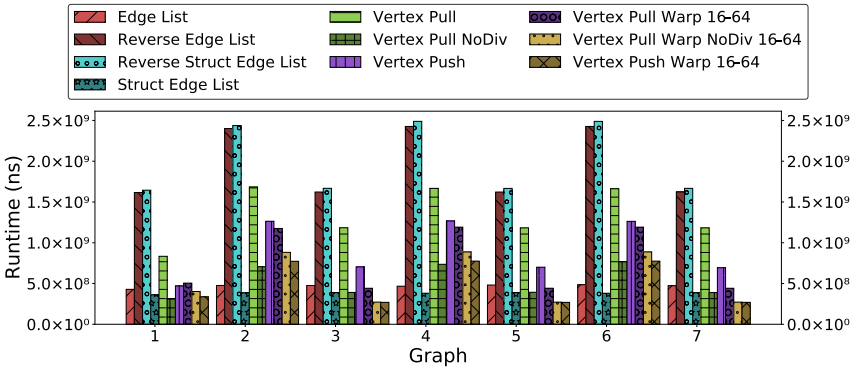


Figure 6.1: Comparison of PageRank run times for different in memory orderings of `actor-collaboration`. See Table 6.1 on the preceding page for the memory ordering of each group.

The answer appears to be: "no". In Fig. 6.1 we show the results of running our implementations on different in-memory orderings of the `actor-collaboration` graph. These reordered version are structurally identical to the original graph, just with the vertices laid out in a different order in memory.

These graphs show performance differences of 2–3× across the different orderings for most implementations. This is surprising, as these different orderings are identical in terms of graph structure and thus total workload as predicted by our workload models.

That the in-memory order of the graphs has such a big impact on the performance demonstrates that the cost of T_{read}, T_{write}, and T_{atom} must depend on this order. This leads us to conclude that we cannot statically approximate of these values, even on a per implementation basis.

6.2.2 Approximation via Performance Counters

While working on the workload models above, we tracked the GPU performance counters during several experiments. Partly to validate our abstract workload models and partly to see if we could find a "smoking gun" that explains the performance differences we saw in Chapter 4 on page 41.

For our edge list implementation these counters correspond beautifully with our concrete workload model. If we work backwards from the performance counter `gld_transactions_per_request`, which represents "32 byte memory accesses per warp", and `gld_transactions`, which tracks "global memory loads", we see the loads for edge list corresponding perfectly with the 7 reads from our concrete workload model for edge list.

Kernel	Order	1	2	3	4	5	6	7
Edge List	Random	$9.1 \cdot 10^6$	9.65	$32.0 \cdot 10^6$	$9.4 \cdot 10^6$	$150.4 \cdot 10^6$	22.85	433
	Abs	$10.2 \cdot 10^6$	10.86	–	–	–	–	449
	Worst	$11.0 \cdot 10^6$	11.72	–	–	–	–	455
Vertex Push	Random	$20.5 \cdot 10^6$	4.52	$21.4 \cdot 10^6$	$9.2 \cdot 10^6$	$43.8 \cdot 10^6$	9.51	449
	Abs	$16.3 \cdot 10^6$	17.33	$7.8 \cdot 10^6$	$2.0 \cdot 10^6$	$16.8 \cdot 10^6$	16.53	1,257
	Worst	$21.4 \cdot 10^6$	2.24	$23.5 \cdot 10^6$	$19.2 \cdot 10^6$	$48.1 \cdot 10^6$	5.00	693

Table 6.2: Relevant performance counter values for PageRank on the `actor-collaboration` graph, column legend in Table 6.3.

Unfortunately, this is hard to repeat for the other implementations. Most of the performance counters aggregate operations at the warp level, which makes it impossible to get detailed insight into what is going on when kernels exhibit divergence within a warp.

As part of our performance counter investigation, we tried to predict the best performing implementation from just the performance counters. This is not directly applicable without a way to predict how the counters behave for our implementations. However, it lets us explore how well we understand the dynamic behaviour of the GPU if we cheat and assume we already have performance counters.

What we find is that — even with this "perfect" run time information — it is impossible to relate the GPU behaviour to wall-clock time. In Table 6.2 we show memory related performance counters and timings for 2 kernels on 3 different memory orderings of the `actor-collaboration` graph.

The "random" ordering corresponds to the order of the data after conversion from the original files to our format, with no other processing. The "abs" ordering has vertices ordered by their absolute degree — i.e., incoming and outgoing edges — ensuring that vertices within a warp have the same or similar degrees.

Finally, the "worst" ordering takes the previous absolute degree ordering and pairs vertices so that every warp has both one of the highest and one of the lowest degree vertices, maximising divergence in each warp.

In Table 6.2 we see that the memory ordering of the graph has almost no impact on the performance counters of the edge list implementation and the actual run times are roughly the same. Meanwhile, we see that vertex push on the random order graph has the same run time as edge list, despite wildly different values for the performance counters.

We see that vertex push on the random ordering has roughly double the number of atomic operations, with half the atomic coalescing; it also does two-thirds less global loads, but with half the coalescing. While having the same run time as the edge list implementation. We do not have any

Index	Performance Counter
1	`atomic_transactions`
2	`atomic_transactions_per_request`
3	`ldst_issued`
4	`ldst_executed`
5	`gld_transactions`
6	`gld_transactions_per_request`
7	Time (in `ms`)

Table 6.3: Legend for performance counter columns in Table 6.2.

estimated costs for these counters, so it is possible that these changes just happen to trade off equally, but that seems unlikely.

Things get even messier when we start considering the vertex push run on the "abs" sorted version of the graph. We see that for each of the performance counters the run on the "abs" sorted graph performs significantly less memory accesses, while having significantly higher coalescing. However, despite every performance counter indicating that this run performs fewer operations and performs these operations more efficiently, we see that it has nearly triple the run time as the random run.

This is a recurring event when we investigate the performance counters of different implementations and runs. Sometimes the fastest implementation is the one that does the least work and has the highest efficiency. In other cases we see that an implementation/run that is objectively worse according to all the performance counters we track, still ends up being the fastest.

As a result, even when we have "perfect" information about the run time behaviour of our implementations, in the form of the actual performance counters, we are *still* unable to predict the best performing implementation based on this information.

6.3 Conclusion

In this chapter we set out to create an analytical model that allows to predict the best performing implementation of a graph algorithm — PageRank in this case — purely from the structural properties of the input graph.

We created abstract workload models for the pseudocode implementations in Section 4.4.1 on page 51 and show that the deviation of these abstract models from the actual generated PTX assembly is limited to small constant factors.

We see that the workload models, without parallelism taken into account, do not accurately reflect the observed wall-clock times for our im-

plementations. This is most clearly demonstrated by the edge list implementation. The edge list workload model shows that it always performs more work than the other implementations, yet it is one of the fastest PageRank implementations overall.

We show that there is no way statically approximate values for T_{read}, T_{write}, and T_{atom}. Additionally, we show these values cannot be statically approximated per implementation either, as reordering the memory layout of significantly alters the wall-clock time.

Further analysis of the performance counters shows that, even when we have perfect information about the performance counters of each implementation, we are *still* unable to correctly predict which of these implementations is the fastest.

It is *theoretically* possible to account for all the dynamic scheduling, coalescing, and caching behaviour of the GPU. However, doing so requires black box reverse engineering of the exact dynamic behaviour of the GPUs as their chipset designs are generally not made public by manufacturers. So we are forced to conclude that a successful analytical performance model is infeasible.

CHAPTER 7

Data-driven Performance Modelling

In Chapter 4 on page 41 we showed that our Graphical Processing Unit (GPU) implementations of graph algorithms exhibit significant performance differences across input graphs. However, showing that graphs have different structure and different performance, is not sufficient to attribute all or most that performance difference to the difference in structure.

In Chapter 6 on page 85 we attempted to create an analytical model to establish a direct link between graph structure and performance. We introduced an analytical workload model that maps input graphs to a number of memory accesses.

We showed that relating these memory accesses to parallel execution time is infeasible, as it depends on unknown details of the hardware in use. Conceptually, it is possible to reverse engineer these hardware details via microbenchmarking, but this is time consuming and specific to a single hardware platform. Our aim is to model the performance impact of graph structure without becoming overly hardware specific.

This chapter is based on work previously presented in:
Merijn Verstraaten et al. *Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach.* 2017. eprint: arXiv:1708.01159
Merijn Verstraaten et al. "Mix-and-Match: A Model-driven Runtime Optimisation Strategy for BFS on GPUs". In: *Proceedings of the 8th Workshop on Irregular Applications: Architectures and Algorithms.* IEEE. 2018, pp. 53–60

In this chapter we present a data-driven approach to performance modelling GPU graph algorithms, based on Binary Decision Trees (BDTs). First, we provide an overview of how BDTs work, what their strong and weak points are, and what makes them suitable for our problem.

We then demonstrate the effectiveness of our BDT based modelling on Breadth-First Search (BFS). Selecting the right implementation at every BFS level provides a speedup of roughly 2×. However, there are overheads associated with switching implementations at run time. We built a prototype BFS traversal to see how big this overhead is. This BFS traversal dynamically switches implementations based on our BDT models.

We show that, in a limited comparison, our adaptive BFS outperforms two state-of-the-art GPU graph processing systems. Our performance improves up to 1.8× over Gunrock [97] and more than 40× over LonestarGPU 2.0 [17].

7.1 Binary Decision Trees

Decision trees are a non-parametric, supervised learning technique [11]. They come in two flavours, *classifiers* and *regressors*, used for categorisation and numerical approximation respectively.

The starting point for the construction of a decision tree is a *training set* of (X, Y) pairs. Here, X is a tuple of 1 or more inputs to the decision tree predictor and Y is a tuple of 1 or more output values we wish to predict. The working assumption is that the original learning set is representative of all observable input/output pairs.

To construct a (binary) decision tree we recursively partition the training set along one of the N input parameters, preferring the parameter (and value) that has the strongest discriminating power — i.e., the one that produces a partitioning closest to 50–50. We assign all elements where the parameter is less than our threshold to the left branch and the others to the right branch. We repeat this process until we reach a stopping criterion — e.g., the maximum tree height, minimal bucket size, etc.

When we reach the stop condition, we compute the predicted output for each leaf node. If all elements of a leaf node have the same value, the prediction is simply that value. However, it is possible for a leaf node to have multiple different output values. In this case the values need to be converted to a single output.

For regressions this usually done by averaging all values in the bucket. For classification, this is usually by selecting the "most likely" value in the bucket, although more complicated strategies exist.

Computing the prediction for an input tuple t is a matter of walking down the tree's structure. At each level we compare the values in t against

the parameter threshold stored in the tree, and recurse down the correct branch. Once we reach a leaf node, we have the prediction for t.

Different decision tree algorithms use different methods to compute which parameters and values are used to split the tree. We used the BDT implementation in the Python library `scikit-learn` [75]. This library uses an algorithm based on the Classification And Regression Trees (CART) [11] algorithm.

This algorithm splits the training set based on which parameter produces the largest reduction in Gini impurity. Gini impurity is a measure of how often an element in a subset would be labelled wrong if all elements in the subset were labelled randomly, according to the distribution of labels in that subset. In other words, it is a measure of how much the variation in subsets is reduced.

Due to the way trees are constructed, overfitting issues can become more pronounced if the input parameters in the learning set are not uniformly distributed across the range we intend to predict against. Additionally, as the number of input parameters increases it becomes exponentially more costly to compute the best discriminator, which in turn makes the algorithm slower and increases the risk of bias and overfitting.

The main advantage of BDTs over newer, more popular, machine learning approaches — such as Support Vector Machines (SVMs), Convolutional Neural Networks (CNNs), and deep learning — is the fact that BDTs are not black box models. The resulting models can be inspected and analysed. One of the main problems with black box machine learning is that it is never clear what, exactly, the model learned, so they cannot help refine our analytical understanding.

For example, CART BDTs let us compute the importance of each input parameter. We can estimate this importance by computing the Gini importance. The Gini importance of a parameter is the total, normalised reduction of Gini impurity by all the splits on that parameter.

Similar importance measures exist for other decision tree algorithms, meaning that we can relatively easily compute the importance of each parameter in our input. This lets us analyse which of our training parameters have the biggest impact and can help inform future analytical modelling.

To summarise the advantages of decision trees:

- They are simple to understand and interpret;
- Small trees can be visualised;
- They require little to no data preparation;
- Prediction cost is logarithmic in the number of data points used;
- They can handle both categorical and numerical data;

- Parameter importance is known after training.

The main downsides of decision trees are:

- Small differences in data can result in drastically different results (i.e., unstable models);
- Constructing optimal decision trees is NP-complete under several aspects of optimality;
- They cannot represent all concepts easily (XOR, parity, multiplexer problems);
- They are prone to overfitting if the training set is biased;
- Biased trees are easily created if some classes dominate.

7.2 Modelling BFS Performance

In this section we elaborate how our BDT training and evaluation process integrates with our toolchain from Chapter 3 on page 17. In Section 3.3.3 on page 35 we explained how we store the metadata needed to train machine learning models. Including our seeded Pseudo Random Number Generator (PRNG)-based approach to generating training and validation sets, allowing us to reconstruct past training and validation sets from a single seed and the model specific configuration.

Figure 7.1 on the next page shows an extended version of our high level schema, including BDT-specific training metadata. In addition to the data covered in Section 3.3.3 on page 35, we store the following model metadata:

- The feature importances of the properties used by the model, and
- the unknown or ambiguous predictions in the model.

The former are useful for informing future analytical modelling efforts and considering what information is important for our models. The latter refers to the information about leaves in our BDT that do not have a single prediction results (see Section 7.1 on page 100). By storing detailed information about these ambiguous leaves we can experiment with different strategies for refining ambiguous predictions.

We mainly focus on the graph size and degree distribution, as our experience and our efforts in Chapter 6 on page 85 lead us to believe that these factors have the biggest impact when it comes to BFS traversals.

Additionally, work on adaptive BFS [9, 58, 65] and run time changes across levels in our dataset [90], indicate that the behaviour at each level

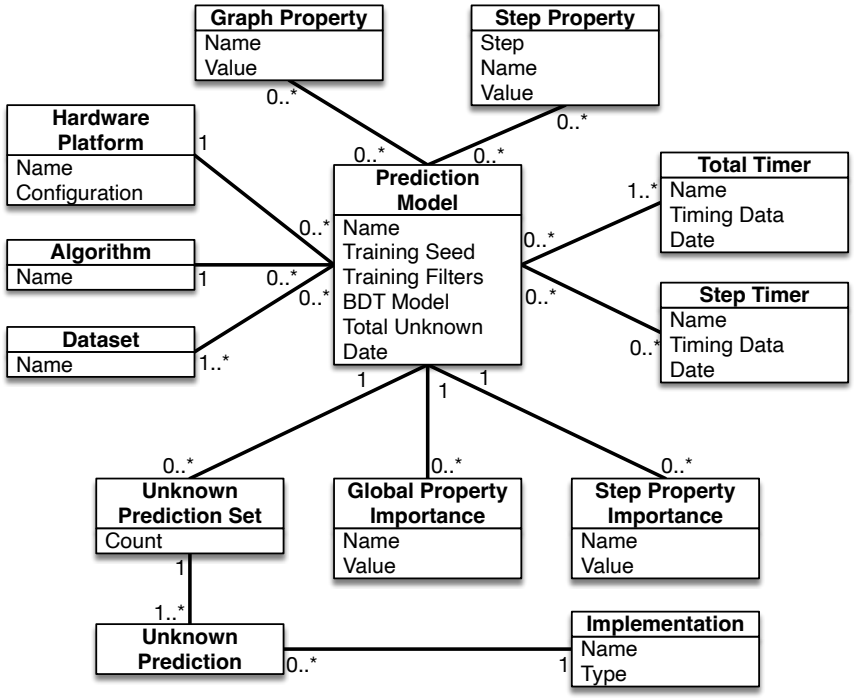


Figure 7.1: BDT specific extension of the schema from Fig. 3.8 on page 36.

correlates with the size of the BFS frontier and the percentage of the graph that has already been explored.

Therefore, we consider the following relevant features for our model:

Graph size:
the number of vertices and edges in the graph.

Frontier size:
either as absolute number of vertices or as percentage of the graph's vertices.

Discovered vertex count:
either as absolute number of vertices or as percentage of the graph's vertices.

Degree distribution:
represented by the five-number summary[1], mean, and standard deviation of the in, out, or absolute degrees of vertices.

[1] That is, the minimum, lower quartile, median, upper quartile, and maximum

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Mix-and-Match (Predicted)	1.74×	2.16×	97%	1%	1%	1344.17×
Best Non-switching	3.51×	2.42×	64%	10%	0%	38.89×
Vertex Push Warp 16–64	7.54×	9.94×	18%	30%	11%	233.97×
Edge List	14.13×	3.73×	52%	22%	2%	53.65×
Vertex Pull Warp 16–64	16.27×	20.13×	5%	64%	19%	1322.53×
Struct Edge List	17.03×	4.27×	49%	25%	2%	67.56×
Reverse Edge List	27.34×	6.39×	42%	32%	7%	97.53×
Vertex Push	27.87×	52.42×	28%	50%	26%	1394.00×
Reverse Struct Edge List	29.34×	6.75×	40%	34%	8%	112.50×
Vertex Pull	37.17×	46.79×	22%	57%	29%	2509.21×

Table 7.1: Predicted Mix-and-Match performance compared to our BFS implementations and the theoretical optimum over all KONECT graphs. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

We base our BDT modelling on the performance data collected in Chapter 4 on page 41 [90]. We ran our 56 BFS implementations and variations on each of the 247 graphs from KONECT [51]; for each of these graphs we performed traversals starting from 10 different starting vertices, and collected the features and run times for each traversed level.

This dataset lets us determine the fastest implementation for each BFS level of each graph. From this we built a dataset where we associate every BFS level with the structural properties of the corresponding graph, the level specific properties, and the fastest implementation. We then train a BDT that predicts the fastest implementation from those level and structural properties.

7.3 Model Evaluation

For our proof-of-concept BDT model we use every feature of the previous section. We train our model on a uniform random selection of 70% of the data points. The first step in evaluating our model's effectiveness is determining whether the predicted implementation switching improves BFS performance. If so, is this improvement significant?

For the evaluations and comparisons in this chapter we use the approach as described in Section 4.3 on page 45. As in Section 4.5 on page 58, we use the term "Best Non-switching" implementation to refer to the performance we would get if we had an oracle to pick the fastest non-switching implementation for each graph in our dataset.

In Table 7.1 we compare the performance of our BDT model, the oracle-

based best non-switching implementation, and the different implementations from Section 4.5 on page 58 against the optimal run time. Over the entire set of KONECT graphs our BDT model's predictions lead to a total run time of 1.74× of optimal — effectively, a 74% slowdown.

This does not sound great, but it is considerably better than the best non-switching implementation, which has a total run time of 3.51× of optimal, a whopping 251% slowdown. In other words, our model can obtain a speedup of 2× over the best non-switching implementation. In practice, the gain is even more significant, because *there is no known model or oracle for selecting the fastest non-switching implementation.*

7.3.1 Dynamic BFS Challenges

From the previous section we conclude that there are considerable performance gains to be had by switching between BFS implementations. However, there are two hurdles between the *predicted* performance gains above and the performance of a real world switching BFS implementation.

The first issue is the overhead introduced by predicting which implementation to use at every BFS level. Fortunately, BDTs are straightforward to implement efficiently, which was one of the reasons we settled on using them.

We tested our BDT implementation against our entire dataset. Predictions were computed in, on average, about 100–200 nanoseconds. This overhead is negligible, as the fastest kernel execution we observed in the dataset takes 20 microseconds to complete.

The second and more fundamental problem is the in-memory representation of input graphs. Most of our implementations operate on different representations. Thus, switching between implementations involves switching between their in-memory representations of the graph. To do so, we need to either: (1) bring new representations into GPU memory on-the-fly, or (2) keep all representations in memory simultaneously.

We considered option (1) infeasible, as transferring data to and from the GPU is slow; doing so repeatedly would be prohibitively expensive. Instead, we chose to consider this as a classical time-space trade-off, where we trade memory for faster compute time by keeping each necessary representation of the graph in memory.

We estimate that the memory overhead introduced by this approach is manageable. The two main graph representations we use are Compressed Sparse Row (CSR) for the vertex-centric implementations and edge list for the edge-centric implementations.

We can combine these two by simply storing the origin vertex for every edge in our CSR. This increases the storage from 1 `int` per vertex and 1 `int` per edge (for CSR) and 2 `int` per edge (for edge list), to 1 `int` per

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Mix-and-Match (Predicted)	1.74×	2.16×	97%	1%	1%	1344.17×
Mix-and-Match (Actual)	2.21×	2.61×	90%	3%	1%	1338.80×
Best Non-switching	3.51×	2.42×	64%	10%	0%	38.89×
Vertex Push Warp 16–64	7.54×	9.94×	18%	30%	11%	233.97×
Edge List	14.13×	3.73×	52%	22%	2%	53.65×
Vertex Pull Warp 16–64	16.27×	20.13×	5%	64%	19%	1322.53×
Struct Edge List	17.03×	4.27×	49%	25%	2%	67.56×
Reverse Edge List	27.34×	6.39×	42%	32%	7%	97.53×
Vertex Push	27.87×	52.42×	28%	50%	26%	1394.00×
Reverse Struct Edge List	29.34×	6.75×	40%	34%	8%	112.50×
Vertex Pull	37.17×	46.79×	22%	57%	29%	2509.21×

Table 7.2: Actual Mix-and-Match performance compared to our BFS implementations and the theoretical optimum over all KONECT graphs. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

vertex and 2 `int` per edge. Assuming a 4 byte `int`, this boils down to a mere 38 MiB for a graph of 10,000,000 edges.

Accommodating the reversed representations too doubles the required space, but given the memory sizes available on modern GPUs this does not seem too problematic. We conclude that both the prediction overhead and extra space requirements for a dynamically switching BFS can be overcome without too much effort.

7.3.2 Mix-and-Match Performance

We validated the above conclusion that prediction and memory overhead can be overcome by implementing "Mix-and-Match", an adaptive BFS implementation. Mix-and-Match can switch between our BFS implementations based on BDT predictions.

We measured the performance of our Mix-and-Match BFS across all of the KONECT graphs. The results of these experiments are shown in Table 7.2. We see that there is non-trivial overhead to switching implementations at run time. Our model predicts run time of 1.74× of optimal, while our Mix-and-Match implementation using that model only achieves a run time of 2.21× of optimal.

Back in Section 4.3.2 on page 47 we discussed that a tiny percentage of our measurements have extremely large Relative Standard Errors (RSEs) and that a large portion of those measurements relate to the dynamic implementations from this chapter. Upon examining the measurements

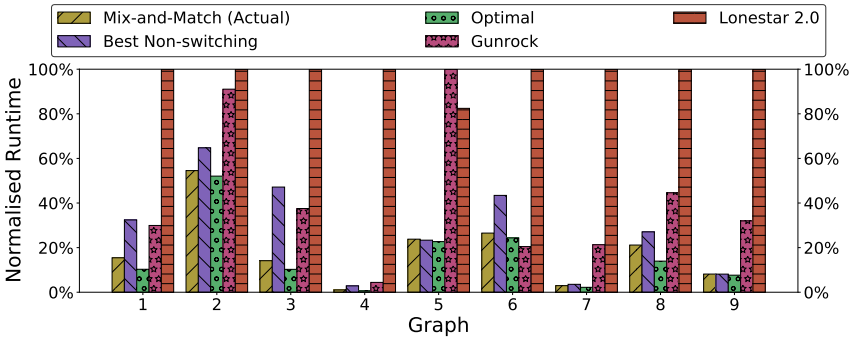


Figure 7.2: Comparison of run times of our optimal baseline, non-switching best, Mix-and-Match, Gunrock, and LonestarGPU 2.0 BFS implementations.

for this chapter we found that the maximum timings were up to 2 orders of magnitude larger than the averages, resulting in equally large RSEs.

So, why is the maximum timing so much larger than the average? Our Mix-and-Match implementation uses a BDT model that is stored in a separate dynamic library. Since we made no efforts to ensure a hot cache it is likely that our BDT model is not loaded into memory until the first prediction is made. Accessing the disk in a critical, timed section would explain the giant outlier and resulting large errors for our dynamic implementations.

The main effect is that the results of our Mix-and-Match implementation look worse than they would under optimal conditions. We opted to keep these measurements as a more accurate reflection of "real world" speedup. As, despite this handicap, our Mix-and-Match implementation is still 1.6× faster than our best non-switching oracle and 3.4× faster than the fastest overall implementation. These results show that our model leads to considerable speedup compared to our individual implementations.

However, speedup results are only as good as our baseline. To see where we stand, We compare our optimal baseline, the best non-switching oracle, and Mix-and-Match against two existing, state-of-the-art GPU graph processing frameworks: Gunrock [97] and LonestarGPU 2.0 [17].

In Fig. 7.2 we show how these implementations compare against the graph selection from Table 4.3 on page 49 in Section 4.3.3 on page 49.

In Table 7.3 on the next page we compare Mix-and-Match against Gunrock and LonestarGPU 2.0 across a random selection of 148 different KONECT graphs. Over this entire dataset, Gunrock achieves of 2.66× of our "optimal". LonestarGPU 2.0 manages 61.36× of optimal. Mix-and-Match only takes 1.44× of optimal, meaning that we are, overall, 1.8× faster than Gunrock and over 40× faster than LonestarGPU 2.0.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Mix-and-Match (Actual)	1.44×	1.54×	91%	1%	0%	17.90×
Gunrock	2.66×	6.17×	9%	48%	1%	25.82×
Best Non-switching	3.64×	2.81×	47%	11%	0%	11.16×
Lonestar 2.0	61.36×	62.13×	5%	80%	43%	1449.72×

Table 7.3: Comparison of BFS performance for Mix-and-Match, Gunrock, and LonestarGPU 2.0 over all KONECT graphs. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

Generality and portability of the model are two big concerns for our model. The above results demonstrate that we can improve the performance of our BFS traversals, both in theory and in practice. However, the existence of one model that improves performance on one dataset is not a compelling argument that our model is more general and applies to multiple datasets.

BDTs perform best when the inputs are uniformly sampled from the parameter space and are known to become biased by bias in the input data. So the generality of our model strongly depends on how representative the KONECT graphs are for real world applications. We will address these issues in more detail in Chapter 8 on page 111.

7.4 Related Work

There have been many advances in large-scale graph traversal algorithms, such as dynamic implementation selection [65], direction switching BFS [9], distributed memory BFS [14], and matrix based graph processing solutions [16]. However, there's still no single best BFS traversal implementation.

This is a result of the input dependence of BFS, with different implementations suffering from different bottlenecks. When combined with complex, massive parallel machines like the GPUs [15], the performance gaps are even more difficult to predict.

Instead of adding yet another BFS implementation, we focus on extracting the best possible performance out of an existing set of implementations by dynamically selecting implementations for each level of a traversal.

This is similar to [58], but our approach combines more algorithms and uses a more deterministic, systematic switching criterion. Moreover, our approach can be extended to incorporate additional BFS versions, as long as sufficient performance data are available for training.

Our Mix-and-Match BFS relies heavily on machine learning based performance prediction. Performance prediction based on machine learning models has been successful in the past [48, 54, 62, 80, 99, 101], but using machine learning *online* to dynamically switch implementations introduces several challenges.

Using machine learning in an online setting means we have to be careful that the overhead of collecting features and computing predictions does not cost more time than we gain. There are also engineering concerns, as we need to integrate the feature collection, prediction, switching to different implementations, and accommodate implementations that work on different data structure. To the best of our knowledge, we are the first to have successfully done this in a generic way that can be extended with any BFS implementation and any BDT model.

7.5 Conclusion

In this chapter we proposed taking a data-driven approach to modelling the performance of GPU graph processing algorithms. We use Binary Decision Trees (BDTs), because they offer a good balance between accuracy and prediction speed.

Although BDT models do not provide any direct insight into the correlations between graph properties and performance, we can still analyse them for insights as BDTs are not a black box method. For example, we can extract the importance of individual properties to from our models, which helps determine the most important properties to consider in further modelling.

We demonstrate the validity of our approach by training a BDT on the BFS performance data gathered in Section 4.5 on page 58 and showing that our BDT model can be used to improve the performance of BFS by dynamically switching between implementations to accommodate the input data.

We implemented Mix-and-Match, an adaptive BFS implementation that can switch between on our existing BFS implementations based on predictions from a BDT. Our experiments demonstrate that, despite the introduced overhead Mix-and-Match is considerably faster than the alternatives.

Our Mix-and-Match is within 1–2× of our optimal baseline in ∼90% of cases, and outperforms our (fictitious) best non-switching oracle by 1.6×. We also compare Mix-and-Match against Gunrock [97] and LonestarGPU 2.0 [17], two state-of-the-art GPU graph processing frameworks. Mix-and-Match outperforms both with an average gain of 1.8× over Gunrock and over 40× for LonestarGPU 2.0. We make no portability claims or guarantees of the Mix-and-Match model presented in this chapter. We

will address these generality and portability concerns of our models in Chapter 8 on the facing page.

Instead, we want to focus on the training and prediction *processes* used. These are systematic, straightforward, and generic, and can be easily applied again different training data, hardware, or algorithms in future studies. We conclude that this systematic benchmarking and modelling approach is a step forward in exploiting the knowledge of structural graph properties for better performance.

CHAPTER 8

Model Portability

In Chapter 7 on page 99 we used our Mix-and-Match Breadth-First Search (BFS) to demonstrate that we can improve the performance of BFS by using Binary Decision Trees (BDTs) to predict the best performing implementation for different stages of a BFS traversal. But this is not enough to defend more general claims we made earlier in this thesis.

Specifically, in Chapters 1, 4, and 6 on page 1, on page 41, and on page 85 we make the following claims:

1. Performance of Graphical Processing Unit (GPU) graph processing is a function of structural properties of the graph, the algorithm being computed, and the hardware.

2. The Single Instruction, Multiple Threads (SIMT) and Stream Processor (SP) based design of General Processing on GPUs (GPGPUs) has not fundamentally changed in the past decade and seems unlikely to be abandoned in the near future.

3. Only a few parallelisation strategies for graph processing can be implemented efficiently on the current SIMT GPGPU architectures.

From the above claims we expect to be able to predict the best GPGPU implementation for an algorithm, based only on structural properties of the graph. The Mix-and-Match BFS from Chapter 7 on page 99 is a proof of concept for BDT-based adaptive BFS. It shows that we can exploit information about a dataset to speed up BFS on that dataset and demonstrates the feasibility of online switching between implementations.

However, Mix-and-Match does *not* demonstrate that:

1. Our BDT models predict the relation between graph structure and performance, as opposed to simply memorising the results for our dataset.

2. That this relation between graph structure and performance is stable across GPU architectures.

In this chapter we address these issues. We demonstrate that the BDT models do not simply memorise the dataset they are trained on. Showing that the models perform well when trained small subsets of our data (Section 8.1) and when used to do predictions on a completely separate dataset (Section 8.2 on page 115).

We also show that the relation between structure and performance is stable across multiple GPU architectures by using our models to do predictions against runs on multiple different generations and architectures of NVIDIA GPUs (Section 8.3 on page 119).

The models and comparisons in this chapter are based on the experiments from Chapter 4 on page 41. For brevity's sake the tables and graphs in this chapter will only include a subset of our implementations. The GPU implementations for these experiments [89], full results of the experiments [90], and code to generate the tables and plots in this chapter [89] are digitally available for interested readers.

8.1 Model Accuracy by Amount of Training Data

We train BDT models for both BFS and PageRank on subsets of our dataset [90] and compare how the performance of models is impacted by the amount of training data.

8.1.1 Breadth-First Search

Our BFS data includes 10 different variants[1] for 247 graphs from the KONECT [51] graph repository. For every level of these BFS variants we have the average performance of each of our implementations.

Since there are multiple different variants for each graph in the dataset there are a number of different ways to select training data for our BDT models:

1. Uniform random subset of graphs,

2. uniform random subset of variants, and

[1] Each variant starts the BFS from a different starting vertex.

3. uniform random subset of steps.

Different selection methods result in different potential biases in the data and resulting model. Training against a random subset of the graphs ensures that all different variants of a single graph are seen. We expect this makes it the impact of the dynamic properties of each BFS level more pronounced and the impact of the static graph properties less pronounced.

Training against a random subset of variants ensures that data from more different graphs is seen, but the model does not see all the different variations in a graph. We expect this reduces the risk of overfitting to a specific set of graphs in the dataset.

Training against a random subset of all steps ensures the model has a *truly* uniform random view of the data. This should eliminate any risk of overfitting to specific graphs, but may not be as effective in learning the impact of dynamic properties, since the model may end up not seeing data from different variants on the same graph at all.

For each of these selections methods we train models on training sets ranging from 10% to 100% of the data using 10% increments, resulting in 30 different BDT models for BFS. Since not all graphs and variants have the same number of steps the percentages of the graph and variant selection methods do not directly correspond to percentages of the total data.

If our BDT models are simply learning the entire dataset we would expect the quality of models to drop rapidly as the size of training data decreases. On the other hand, if the models are actually learning the relation between graph structure and implementation performance, we would expect the model performance to have a more gradual and slow deterioration as the training data decreases.

In Table 8.1 on the following page we compare these 30 models with each other and a selection of non-adaptive implementations, using the approach described in Section 4.3 on page 45. The entries in this table are sorted by their performance across the entire KONECT dataset.

We also include data for "best non-switching" for comparison. This entry mimics the result of having an oracle predicting the best non-adaptive BFS implementation for each variant. Such an oracle does not exist, but it gives a sense for the best attainable result without an adaptive BFS.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (90% Steps)	1.28×	1.37×	98%	1%	0%	335.38×
KONECT on TitanX (100% Graphs)	1.43×	1.47×	98%	1%	0%	412.03×
KONECT on TitanX (90% Variants)	1.47×	1.57×	97%	1%	0%	412.03×
KONECT on TitanX (100% Steps)	1.48×	1.47×	98%	1%	0%	412.03×
KONECT on TitanX (100% Variants)	1.50×	1.48×	98%	1%	0%	412.03×
KONECT on TitanX (80% Variants)	1.57×	1.62×	97%	1%	0%	412.05×
KONECT on TitanX (80% Steps)	1.59×	1.61×	97%	1%	0%	412.03×

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (70% Variants)	1.65×	1.87×	96%	1%	0%	448.61×
KONECT on TitanX (70% Steps)	1.74×	2.16×	97%	1%	1%	1344.17×
KONECT on TitanX (60% Variants)	1.75×	2.19×	95%	2%	1%	950.48×
KONECT on TitanX (40% Variants)	1.79×	2.06×	93%	3%	1%	336.01×
KONECT on TitanX (60% Steps)	1.80×	1.90×	97%	1%	1%	482.20×
KONECT on TitanX (80% Graphs)	1.92×	3.06×	92%	2%	1%	430.42×
KONECT on TitanX (90% Graphs)	2.01×	3.01×	94%	2%	1%	610.80×
KONECT on TitanX (50% Steps)	2.04×	2.14×	94%	3%	1%	335.38×
KONECT on TitanX (30% Variants)	2.39×	3.06×	91%	4%	1%	567.48×
KONECT on TitanX (70% Graphs)	2.53×	5.27×	91%	4%	2%	854.48×
KONECT on TitanX (50% Variants)	2.62×	2.50×	93%	3%	1%	454.23×
KONECT on TitanX (40% Steps)	2.76×	2.91×	92%	4%	1%	561.52×
KONECT on TitanX (20% Steps)	2.86×	3.91×	86%	8%	3%	749.13×
KONECT on TitanX (60% Graphs)	3.08×	6.22×	89%	5%	3%	857.67×
KONECT on TitanX (50% Graphs)	3.10×	5.97×	87%	4%	2%	847.82×
KONECT on TitanX (20% Variants)	3.16×	6.74×	87%	7%	3%	763.81×
KONECT on TitanX (30% Steps)	3.19×	3.11×	91%	5%	2%	925.58×
KONECT on TitanX (40% Graphs)	3.50×	6.85×	84%	7%	2%	856.27×
Best Non-switching	3.51×	2.42×	64%	10%	0%	38.89×
KONECT on TitanX (10% Variants)	3.51×	5.31×	81%	9%	3%	856.28×
KONECT on TitanX (30% Graphs)	3.61×	6.80×	76%	10%	3%	856.42×
KONECT on TitanX (10% Steps)	3.95×	5.39×	82%	8%	3%	879.75×
Vertex Push Warp 16–64	7.54×	9.94×	18%	30%	11%	233.97×
KONECT on TitanX (20% Graphs)	7.72×	9.07×	75%	14%	5%	457.57×
KONECT on TitanX (10% Graphs)	7.81×	24.41×	57%	22%	11%	1344.36×
Edge List	14.13×	3.73×	52%	22%	2%	53.65×
Vertex Pull Warp 16–64	16.27×	20.13×	5%	64%	19%	1322.53×
Struct Edge List	17.03×	4.27×	49%	25%	2%	67.56×
Reverse Edge List	27.34×	6.39×	42%	32%	7%	97.53×
Vertex Push	27.87×	52.42×	28%	50%	26%	1394.00×
Reverse Struct Edge List	29.34×	6.75×	40%	34%	8%	112.50×
Vertex Pull	37.17×	46.79×	22%	57%	29%	2509.21×

Table 8.1: Aggregate performance of BDT models trained on different training set sizes compared to non-switching BFS implementations. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

Models trained using the graph based selection strategy perform considerably worse than the other two training strategies. This matches our intuition and expectations, although the effect is more pronounced than expected. The other two selection strategies, step and variant based, perform comparably well. Models trained on as little as 30% of the dataset already outperform both our (non-existent) best non-switching oracle and the best non-adaptive implementation we have.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (100% Graphs)	1.01×	1.00×	100%	0%	0%	1.16×
KONECT on TitanX (90% Graphs)	1.02×	1.01×	100%	0%	0%	2.24×
KONECT on TitanX (60% Graphs)	1.02×	1.04×	99%	0%	0%	2.29×
KONECT on TitanX (70% Graphs)	1.05×	1.05×	98%	0%	0%	4.09×
KONECT on TitanX (50% Graphs)	1.08×	1.05×	100%	0%	0%	6.14×
KONECT on TitanX (30% Graphs)	1.08×	1.12×	98%	1%	0%	13.11×
KONECT on TitanX (20% Graphs)	1.08×	1.59×	86%	4%	0%	15.58×
KONECT on TitanX (10% Graphs)	1.09×	1.50×	90%	4%	0%	15.58×
KONECT on TitanX (40% Graphs)	1.10×	1.07×	98%	0%	0%	6.14×
Struct Edge List	1.13×	1.07×	99%	0%	0%	2.58×
KONECT on TitanX (80% Graphs)	1.13×	1.06×	99%	0%	0%	7.89×
Edge List	1.16×	1.14×	99%	0%	0%	2.62×
Vertex Pull NoDiv	2.43×	2.93×	59%	14%	0%	18.80×
Reverse Edge List	2.49×	2.09×	71%	7%	0%	8.81×
Reverse Struct Edge List	2.50×	2.09×	71%	7%	0%	8.85×
Vertex Push Warp 16–64	2.71×	4.75×	37%	19%	4%	66.63×
Vertex Pull Warp NoDiv 16–64	3.51×	5.42×	13%	37%	1%	27.46×
Vertex Pull Warp 16–64	4.74×	6.88×	5%	47%	4%	35.05×
Vertex Push	5.86×	16.66×	41%	34%	12%	642.08×
Vertex Pull	14.59×	16.68×	17%	60%	27%	143.45×

Table 8.2: Comparison of BDT models against PageRank implementations. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

8.1.2 PageRank

Unlike BFS, PageRank does not have different variants per graph nor does it have multiple steps with changing behaviour. This means the three different selection strategies discussed in the previous section are equivalent, so we can limit ourselves to models trained on 10% through 100% of the graphs.

In Table 8.2 we compare these 10 models with each other and other implementations, again using the approach described in Section 4.3 on page 45. We see similar results as before with BFS, with nearly all models outperforming the non-predictive implementations.

8.2 Portability Across Datasets

In the previous section we established that our BDT models work, even when trained on small subsets of the available data. Indicating that these models are not simply memorising the result of our entire data set. In this section we want to establish that the behaviour learned by our models generalises beyond the training dataset. That is, it is not specific to the KONECT results they were trained on.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (60% Steps)	1.06×	1.26×	97%	0%	0%	5.72×
KONECT on TitanX (80% Steps)	1.09×	1.38×	93%	1%	0%	39.25×
KONECT on TitanX (70% Variants)	1.09×	1.33×	96%	1%	0%	13.97×
KONECT on TitanX (80% Variants)	1.10×	1.35×	93%	1%	0%	14.09×
KONECT on TitanX (60% Variants)	1.12×	1.36×	91%	0%	0%	11.04×
KONECT on TitanX (50% Steps)	1.12×	1.34×	95%	1%	0%	39.32×
KONECT on TitanX (50% Variants)	1.13×	1.40×	92%	1%	0%	21.00×
KONECT on TitanX (70% Steps)	1.13×	1.45×	92%	1%	0%	39.25×
Best Non-switching	2.19×	1.81×	83%	5%	0%	37.34×
Vertex Push Warp 16–64	3.80×	3.75×	22%	19%	0%	37.34×
Vertex Push	4.82×	11.01×	33%	58%	12%	134.41×
Edge List	5.57×	2.85×	66%	10%	1%	49.05×
Struct Edge List	6.59×	3.21×	64%	15%	1%	59.99×
Reverse Edge List	7.46×	4.19×	57%	17%	3%	79.07×
Reverse Struct Edge List	8.36×	4.41×	56%	20%	3%	84.25×
Vertex Pull Warp 16–64	16.79×	11.71×	0%	64%	9%	409.26×
Vertex Pull	51.31×	22.46×	12%	59%	13%	796.82×

Table 8.3: KONECT BFS models evaluated against SNAP dataset. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

To do this we take our measurements from the SNAP dataset, feed the prediction parameters to our models trained on KONECT, and use the run time of the predicted implementation. If the model's behaviour is, indeed, general, we expect to see similar results as we do on KONECT (see Tables 8.1 and 8.2 on page 114 and on the previous page). We then do the same in reverse, taking models trained on our SNAP dataset measurements and evaluating them against our KONECT data.

For brevity's sake, we limit ourselves to models trained on 50–80% of the dataset using steps and variants. In the machine learning literature 80–20 and 70–30 splits between training and validation data are common, and training on variants and steps results in a more robust uniform sampling of data. This way we can cover the most realistic training setups without resulting in unnecessarily verbose tables.

8.2.1 KONECT to SNAP

The results for our KONECT BFS models evaluated against SNAP are shown in Table 8.3. As before, all the models outperform the non-switching implementations and our non-switching oracle. Our models performance compared to optimal seems to be even better than KONECT — with the best model achieving 1.06× of optimal on SNAP versus 1.28× on the KONECT dataset.

Table 8.4 on the next page shows similar results for PageRank models,

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (80% Graphs)	1.04×	1.06×	99%	0%	0%	3.48×
KONECT on TitanX (60% Graphs)	1.08×	1.07×	98%	0%	0%	3.48×
KONECT on TitanX (70% Graphs)	1.14×	1.11×	97%	0%	0%	4.10×
Vertex Pull NoDiv	1.42×	2.60×	52%	5%	0%	8.78×
Struct Edge List	1.54×	1.14×	96%	0%	0%	3.48×
Edge List	1.55×	1.21×	96%	0%	0%	3.54×
Vertex Pull Warp NoDiv 16–64	1.69×	5.12×	8%	45%	0%	15.11×
Reverse Edge List	2.20×	1.66×	83%	2%	0%	10.60×
Reverse Struct Edge List	2.25×	1.64×	83%	2%	0%	10.76×
Vertex Push Warp 16–64	2.27×	3.03×	28%	11%	0%	7.47×
Vertex Pull Warp 16–64	3.03×	6.50×	1%	54%	0%	19.10×
Vertex Push	3.08×	6.75×	36%	44%	6%	44.76×
Vertex Pull	4.57×	15.03×	11%	62%	32%	68.48×

Table 8.4: KONECT PageRank models evaluated against SNAP dataset. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

once again showing that our models outperforms the other implementations on SNAP. This supports our argument that the relation between performance and graph structure can be generalised from KONECT to other datasets.

8.2.2 SNAP to KONECT

In Tables 8.5 and 8.6 on the following page we show the reverse of Section 8.2.1 on the preceding page. We take models that were trained on our SNAP result and evaluate them against the KONECT results.

Unfortunately, the results in this direction are less promising. For BFS the models perform considerably worse, with all models performing worse than the best non-switching oracle and the push warp implementation. While the models perform slightly better on PageRank, you are still better off simply always using either of the edge list implementations.

8.2.3 Conclusions on Dataset Portability

One possible explanation for the poorer performance when going from SNAP to KONECT is that (the part of) the SNAP dataset we used has considerably less variation in the types and sizes of graphs it includes. This could result in a model biased towards some classes of graphs, leading to reduced generality and thus poorer performance on the more varied KONECT dataset.

Intuitively, it makes sense that our training dataset(s) need to be "sufficiently varied" to produce general models. But defining "sufficiently varied" is a complex problem. As we have also discussed in Chapter 5 on page 67

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Best Non-switching	3.51×	2.42×	64%	10%	0%	38.89×
Vertex Push Warp 16–64	7.54×	9.94×	18%	30%	11%	233.97×
SNAP on TitanX (80% Steps)	11.03×	10.89×	69%	15%	6%	751.01×
SNAP on TitanX (70% Variants)	13.00×	17.01×	65%	18%	13%	838.19×
Edge List	14.13×	3.73×	52%	22%	2%	53.65×
SNAP on TitanX (60% Variants)	14.54×	16.31×	65%	20%	12%	838.29×
SNAP on TitanX (80% Variants)	15.93×	15.86×	69%	16%	10%	869.29×
Vertex Pull Warp 16–64	16.27×	20.13×	5%	64%	19%	1322.53×
SNAP on TitanX (70% Steps)	16.85×	17.09×	69%	17%	11%	954.19×
Struct Edge List	17.03×	4.27×	49%	25%	2%	67.56×
SNAP on TitanX (60% Steps)	17.04×	16.46×	69%	18%	12%	1093.98×
Reverse Edge List	27.34×	6.39×	42%	32%	7%	97.53×
Vertex Push	27.87×	52.42×	28%	50%	26%	1394.00×
Reverse Struct Edge List	29.34×	6.75×	40%	34%	8%	112.50×
Vertex Pull	37.17×	46.79×	22%	57%	29%	2509.21×

Table 8.5: SNAP BFS models evaluated against KONECT dataset. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
Struct Edge List	1.13×	1.07×	99%	0%	0%	2.58×
Edge List	1.16×	1.14×	99%	0%	0%	2.62×
SNAP on TitanX (80% Graphs)	1.33×	1.21×	95%	2%	0%	6.14×
SNAP on TitanX (70% Graphs)	2.16×	1.41×	91%	3%	0%	7.98×
SNAP on TitanX (60% Graphs)	2.39×	2.06×	84%	9%	0%	18.80×
Vertex Pull NoDiv	2.43×	2.93×	59%	14%	0%	18.80×
Reverse Edge List	2.49×	2.09×	71%	7%	0%	8.81×
Reverse Struct Edge List	2.50×	2.09×	71%	7%	0%	8.85×
Vertex Push Warp 16–64	2.71×	4.75×	37%	19%	4%	66.63×
Vertex Pull Warp NoDiv 16–64	3.51×	5.42×	13%	37%	1%	27.46×
Vertex Pull Warp 16–64	4.74×	6.88×	5%	47%	4%	35.05×
Vertex Push	5.86×	16.66×	41%	34%	12%	642.08×
Vertex Pull	14.59×	16.68×	17%	60%	27%	143.45×

Table 8.6: SNAP PageRank models evaluated against KONECT dataset. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

there is no comprehensive way of classifying graphs, meaning there is also no way to classify the variety of datasets.

This problem could be approached empirically by gathering up many unique graphs, taking the powerset of that collection to create a large number of unique datasets, then comparing the performance of models trained on each of these datasets against all the others.

However, this approach is rather costly and still leaves open the question whether the initial set of graphs was "representative"[2]. We consider this question beyond the scope of this thesis.

The excellent performance of the KONECT models on SNAP do provide evidence that models trained using out workflow *can be* more generally applicable beyond the dataset that they were trained. Additionally, these results strengthen our argument that our BDT models are *not* simply "memorising" the results for the dataset they are trained on.

8.3 Portability Across GPUs

So far we have looked at the data requirements of our BDT models and how portable they are across datasets. However, all this has been in the context of data gathered using a single type of GPU, the NVIDIA TitanX.

At the beginning of this chapter we described how — in earlier chapters — we assume that our models can predict the best implementation based only on the structure of the graph and the implementations. That is, independent of the actual GPU. In this section we will show that this is indeed the case.

We have performed all our BFS measurements on the KONECT dataset on 4 different GPUs from different generations and with different architectures:

NVIDIA K20, a Kepler microarchitecture card from 2012

NVIDIA GTX980, a Maxwell microarchitecture card from 2014

NVIDIA TitanX, a Maxwell microarchitecture card from 2014

NVIDIA RTX2080 Ti, a Turing microarchitecture card from 2018

We trained our BFS models for each of these 4 GPUs. If our hypothesis is correct, we expect our models to predict the "correct" implementation to use for a graph, even if that model was trained on data produced by a different GPU.

In Table 8.7 on the next page we show the results of models trained on each of the 4 GPUs when predicting implementations on the K20 GPU. We

[2] However we choose to define representative…

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (80% Steps)	1.25×	1.68×	97%	1%	0%	384.67×
KONECT on K20 (70% Steps)	1.26×	1.57×	97%	1%	0%	410.27×
KONECT on TitanX (80% Variants)	1.29×	1.71×	96%	1%	0%	384.66×
KONECT on GTX980 (80% Variants)	1.29×	1.61×	96%	1%	0%	384.66×
KONECT on GTX980 (70% Variants)	1.30×	1.54×	97%	1%	0%	384.66×
KONECT on K20 (80% Steps)	1.33×	1.88×	98%	1%	0%	463.57×
KONECT on GTX980 (60% Steps)	1.35×	1.80×	96%	1%	0%	435.84×
KONECT on K20 (70% Variants)	1.35×	1.54×	98%	1%	0%	410.27×
KONECT on K20 (80% Variants)	1.37×	1.52×	98%	1%	0%	410.28×
KONECT on GTX980 (80% Steps)	1.37×	1.67×	97%	1%	0%	435.84×
KONECT on RTX2080Ti (80% Steps)	1.38×	1.94×	97%	1%	0%	595.24×
KONECT on RTX2080Ti (80% Variants)	1.39×	1.87×	97%	1%	0%	443.58×
KONECT on TitanX (60% Steps)	1.40×	1.92×	96%	1%	0%	436.15×
KONECT on GTX980 (70% Steps)	1.40×	1.74×	96%	1%	0%	435.84×
KONECT on RTX2080Ti (70% Variants)	1.41×	1.82×	97%	1%	1%	410.27×
KONECT on TitanX (70% Steps)	1.42×	2.17×	96%	1%	0%	1203.37×
KONECT on RTX2080Ti (60% Steps)	1.44×	2.20×	95%	2%	1%	410.27×
KONECT on RTX2080Ti (70% Steps)	1.47×	1.99×	96%	1%	0%	595.24×
KONECT on TitanX (60% Variants)	1.52×	2.22×	94%	2%	1%	848.46×
KONECT on TitanX (70% Variants)	1.55×	1.95×	95%	2%	0%	411.34×
KONECT on RTX2080Ti (60% Variants)	1.60×	2.57×	96%	2%	1%	415.71×
KONECT on K20 (60% Variants)	1.63×	1.85×	97%	1%	0%	415.41×
KONECT on K20 (60% Steps)	1.69×	1.89×	96%	2%	1%	463.57×
KONECT on GTX980 (60% Variants)	1.81×	2.14×	95%	2%	1%	443.58×
Best Non-switching	2.85×	2.12×	67%	7%	0%	12.37×
Edge List	4.96×	2.74×	58%	15%	0%	45.43×
Struct Edge List	5.35×	2.88×	57%	16%	0%	52.01×
Vertex Push Warp 16–64	7.80×	11.33×	7%	45%	12%	232.17×
Reverse Edge List	11.42×	5.32×	42%	30%	4%	129.94×
Reverse Struct Edge List	11.73×	5.40×	42%	30%	4%	134.00×
Vertex Push	17.97×	47.52×	27%	50%	24%	1248.00×
Vertex Pull Warp 16–64	19.03×	22.30×	2%	85%	21%	1438.64×
Vertex Pull	35.70×	43.71×	19%	56%	27%	2530.27×

Table 8.7: KONECT BFS models against K20. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

see that all the models outperform our best non-switching implementation by a decent margin, regardless of the hardware they were trained on.

We also compare our 4 sets of models against the results of the GTX980 (Table 8.8 on the following page), the results of the TitanX (Table 8.9 on page 123), and the results of the RTX2080 Ti (Table 8.10 on page 124). We see similar results in each of these tables. There is some variation across the different hardware, but in each case the model outperform our non-switching oracle.

This validates our assumption that the architecture of GPUs over the past decade and for the foreseeable future are similar enough that we can attribute most of the observed performance differences to the match between graph structure and parallelisation strategy (see Section 4.2 on page 43).

8.4 Conclusion

In this chapter we set out to demonstrate two things:

1. That our BDT models do not simply memorise results, but predict the relation between graph structure and performance.

2. That this relation between graph structure and performance is stable across GPU architectures.

In Section 8.1 on page 112 we showed that the BDT models perform well, even if they see only small amounts of our dataset. And in Section 8.2 on page 115 we show that the models, to some extent, even work when used with datasets they have not seen at all. Thus, we conclude that the BDT model is, indeed, not simply memorising the training data.

In Section 8.3 on page 119 we showed that the performance of the models is largely independent of the GPU they were trained on. Leading us to conclude that the performance of our BDT models is stable across GPU architectures, as expected.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (80% Variants)	1.61×	1.66×	97%	1%	0%	411.25×
KONECT on TitanX (80% Steps)	1.61×	1.64×	97%	1%	0%	411.23×
KONECT on TitanX (70% Variants)	1.68×	1.92×	96%	1%	0%	463.61×
KONECT on GTX980 (70% Variants)	1.77×	1.53×	97%	1%	0%	411.23×
KONECT on TitanX (70% Steps)	1.79×	2.17×	97%	2%	1%	1268.31×
KONECT on TitanX (60% Variants)	1.82×	2.26×	95%	2%	1%	990.60×
KONECT on TitanX (60% Steps)	1.83×	1.92×	97%	1%	1%	481.89×
KONECT on GTX980 (80% Variants)	1.92×	1.58×	97%	1%	0%	411.23×
KONECT on GTX980 (80% Steps)	1.98×	1.59×	97%	1%	0%	481.79×
KONECT on GTX980 (60% Variants)	2.26×	2.20×	96%	2%	1%	532.47×
KONECT on K20 (60% Steps)	2.27×	2.18×	95%	2%	1%	570.81×
KONECT on GTX980 (60% Steps)	2.43×	1.80×	96%	2%	0%	481.79×
KONECT on GTX980 (70% Steps)	2.52×	1.71×	96%	1%	0%	481.79×
KONECT on RTX2080Ti (60% Steps)	2.56×	2.35×	95%	3%	1%	517.63×
KONECT on RTX2080Ti (80% Steps)	2.61×	2.13×	96%	2%	1%	655.70×
KONECT on RTX2080Ti (70% Steps)	2.65×	2.22×	95%	2%	1%	655.70×
KONECT on K20 (60% Variants)	2.77×	2.16×	95%	2%	1%	517.65×
KONECT on RTX2080Ti (70% Variants)	2.77×	2.07×	95%	2%	1%	517.63×
KONECT on K20 (70% Steps)	2.79×	1.83×	96%	2%	1%	517.63×
KONECT on K20 (80% Steps)	2.80×	2.18×	96%	2%	1%	570.81×
KONECT on K20 (70% Variants)	2.85×	1.82×	96%	2%	1%	517.63×
KONECT on RTX2080Ti (60% Variants)	2.87×	2.83×	94%	3%	1%	517.63×
KONECT on RTX2080Ti (80% Variants)	3.13×	2.12×	96%	2%	1%	532.45×
KONECT on K20 (80% Variants)	3.21×	1.79×	96%	2%	0%	517.65×
Best Non-switching	3.70×	2.36×	66%	11%	0%	42.29×
Vertex Push Warp 16–64	8.01×	10.32×	27%	31%	12%	232.96×
Edge List	11.60×	3.41×	55%	20%	1%	49.82×
Struct Edge List	13.98×	3.91×	49%	23%	2%	61.71×
Vertex Pull Warp 16–64	17.61×	20.84×	7%	60%	19%	1266.57×
Vertex Push	28.98×	52.52×	32%	48%	27%	1394.66×
Reverse Edge List	29.35×	6.51×	44%	30%	7%	123.57×
Reverse Struct Edge List	31.26×	6.93×	43%	32%	8%	137.46×
Vertex Pull	38.50×	48.16×	26%	56%	30%	2402.22×

Table 8.8: KONECT BFS models against GTX980. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (80% Variants)	1.57×	1.62×	97%	1%	0%	412.05×
KONECT on TitanX (80% Steps)	1.59×	1.61×	97%	1%	0%	412.03×
KONECT on TitanX (70% Variants)	1.65×	1.87×	96%	1%	0%	448.61×
KONECT on TitanX (70% Steps)	1.74×	2.16×	97%	1%	1%	1344.17×
KONECT on GTX980 (70% Variants)	1.75×	1.60×	97%	1%	0%	412.03×
KONECT on TitanX (60% Variants)	1.75×	2.19×	95%	2%	1%	950.48×
KONECT on TitanX (60% Steps)	1.80×	1.90×	97%	1%	1%	482.20×
KONECT on GTX980 (80% Variants)	1.85×	1.65×	97%	1%	0%	412.03×
KONECT on GTX980 (80% Steps)	1.95×	1.66×	97%	1%	0%	482.09×
KONECT on K20 (60% Steps)	2.21×	2.16×	95%	2%	1%	545.02×
KONECT on GTX980 (60% Variants)	2.25×	2.23×	96%	2%	1%	525.47×
KONECT on GTX980 (60% Steps)	2.33×	1.87×	96%	2%	0%	482.09×
KONECT on GTX980 (70% Steps)	2.44×	1.75×	96%	1%	0%	482.09×
KONECT on RTX2080Ti (60% Steps)	2.52×	2.37×	94%	3%	1%	475.40×
KONECT on RTX2080Ti (80% Steps)	2.55×	2.16×	95%	2%	1%	653.68×
KONECT on RTX2080Ti (70% Steps)	2.60×	2.24×	95%	2%	1%	653.68×
KONECT on RTX2080Ti (70% Variants)	2.69×	2.09×	95%	2%	1%	475.40×
KONECT on K20 (60% Variants)	2.70×	2.15×	95%	2%	1%	475.41×
KONECT on K20 (70% Steps)	2.71×	1.83×	96%	2%	1%	475.40×
KONECT on K20 (80% Steps)	2.73×	2.18×	96%	2%	1%	545.02×
KONECT on K20 (70% Variants)	2.78×	1.82×	96%	2%	1%	475.40×
KONECT on RTX2080Ti (60% Variants)	2.80×	2.85×	94%	3%	1%	475.40×
KONECT on RTX2080Ti (80% Variants)	3.03×	2.13×	95%	2%	1%	505.04×
KONECT on K20 (80% Variants)	3.14×	1.80×	96%	2%	0%	475.41×
Best Non-switching	3.51×	2.42×	64%	10%	0%	38.89×
Vertex Push Warp 16–64	7.54×	9.94×	18%	30%	11%	233.97×
Edge List	14.13×	3.73×	52%	22%	2%	53.65×
Vertex Pull Warp 16–64	16.27×	20.13×	5%	64%	19%	1322.53×
Struct Edge List	17.03×	4.27×	49%	25%	2%	67.56×
Reverse Edge List	27.34×	6.39×	42%	32%	7%	97.53×
Vertex Push	27.87×	52.42×	28%	50%	26%	1394.00×
Reverse Struct Edge List	29.34×	6.75×	40%	34%	8%	112.50×
Vertex Pull	37.17×	46.79×	22%	57%	29%	2509.21×

Table 8.9: KONECT BFS models against TitanX. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

Algorithm	Total	Avg	1–2×	>5×	>20×	Worst
KONECT on TitanX (70% Steps)	1.59×	2.21×	96%	1%	0%	1173.71×
KONECT on RTX2080Ti (80% Steps)	1.60×	1.92×	98%	1%	1%	600.43×
KONECT on TitanX (60% Steps)	1.60×	1.86×	96%	1%	0%	373.33×
KONECT on RTX2080Ti (60% Steps)	1.66×	2.23×	96%	2%	1%	500.74×
KONECT on TitanX (80% Variants)	1.73×	1.77×	96%	1%	0%	373.24×
KONECT on RTX2080Ti (70% Variants)	1.76×	1.87×	97%	1%	1%	500.74×
KONECT on TitanX (80% Steps)	1.79×	1.70×	97%	1%	0%	373.33×
KONECT on RTX2080Ti (70% Steps)	1.79×	2.00×	97%	1%	1%	600.43×
KONECT on GTX980 (70% Variants)	1.80×	1.66×	97%	1%	0%	373.24×
KONECT on TitanX (70% Variants)	1.85×	1.94×	96%	1%	0%	373.24×
KONECT on K20 (70% Steps)	1.87×	1.75×	97%	1%	0%	500.74×
KONECT on K20 (80% Steps)	1.87×	2.07×	97%	1%	1%	500.74×
KONECT on K20 (60% Variants)	1.88×	2.00×	96%	2%	1%	500.75×
KONECT on GTX980 (80% Variants)	1.88×	1.69×	98%	1%	0%	373.24×
KONECT on RTX2080Ti (60% Variants)	1.88×	2.35×	96%	2%	1%	500.74×
KONECT on K20 (60% Steps)	1.92×	2.06×	96%	2%	1%	500.74×
KONECT on TitanX (60% Variants)	1.92×	2.32×	94%	2%	1%	808.44×
KONECT on K20 (80% Variants)	1.96×	1.68×	97%	1%	0%	500.75×
KONECT on GTX980 (60% Steps)	1.99×	1.83×	97%	1%	0%	286.65×
KONECT on GTX980 (80% Steps)	2.00×	1.58×	98%	1%	0%	300.14×
KONECT on K20 (70% Variants)	2.00×	1.73×	97%	1%	0%	500.74×
KONECT on GTX980 (70% Steps)	2.02×	1.77×	98%	1%	0%	437.74×
KONECT on RTX2080Ti (80% Variants)	2.09×	1.90×	97%	1%	1%	500.74×
KONECT on GTX980 (60% Variants)	2.28×	2.18×	96%	2%	1%	454.09×
Best Non-switching	3.08×	2.06×	71%	9%	0%	18.14×
Edge List	4.22×	2.50×	64%	14%	0%	30.00×
Struct Edge List	4.70×	2.73×	60%	16%	0%	31.11×
Vertex Push Warp 16–64	8.54×	15.60×	16%	37%	15%	572.97×
Reverse Edge List	12.10×	4.57×	56%	24%	3%	62.17×
Reverse Struct Edge List	12.91×	4.87×	53%	26%	4%	70.66×
Vertex Push	19.28×	47.21×	33%	44%	25%	1217.56×
Vertex Pull Warp 16–64	21.96×	31.28×	8%	59%	23%	1543.80×
Vertex Pull	26.98×	34.98×	31%	53%	26%	1704.99×

Table 8.10: KONECT BFS models against RTX2080Ti. See Section 4.3 on page 45 for a detailed explanation on how to read this data and its measurement accuracy.

CHAPTER 9

Conclusion

Performance engineering is the software engineering art of untangling the complex interactions between algorithms, data structures, hardware, and input to achieve the best possible performance. The challenge of performance engineering for General Processing on GPU (GPGPU) graph processing lies at the heart of this thesis.

As mentioned in Chapter 1 on page 1, the cost-effectiveness — both in terms of purchase price and FLOPS per watt — and commodity nature of GPGPUs led to them becoming ubiquitous in High-Performance Computing (HPC) and this will remain the case for the foreseeable future.

The current and continued relevance of GPGPU graph processing follows from the above, combined with the versatility and usefulness of graphs for modelling problems in many fields, such as computational linguistics, physics, biology/bioinformatics, and social science.

9.1 Conclusions

This section shows how the work in this thesis addresses our research hypotheses, as described in Section 1.2 on page 3. In Section 9.1.2 on page 131 we discuss how all the individual parts fit together into a coherent methodology for future exploration of these performance engineering problems. Finally, Section 9.1.3 on page 132 presents a summary of all the contributions in this thesis.

9.1.1 Findings per Research Goal

The findings discussed in the following subsections correspond to the five research objectives enumerated above in Section 1.2 on page 3.

9.1.1.1 Software & Tooling

Performance engineering is a branch of empirical computer science; that is, computer science based on experimental observation. In any empirical science the quality of your results is only as good as your ability to reproduce your results. An aspect that is, unfortunately, still often overlooked in computer science [23].

The experiments within this thesis, despite only addressing a tiny part of graph processing on Graphical Processing Units (GPUs), involve 56 implementations of Breadth-First Search (BFS), 19 implementations of PageRank, 247 graphs from the KONECT dataset, and 109 graphs from the SNAP dataset. Benchmarking all these possible combinations leads to:

$$(56 + 19) \times (247 + 109) = 26,700 \text{ experiments}$$

The above 26,700 experiments is *before* we even consider repeating these experiments across multiple GPU generations or the fact that BFS has different behaviour for every possible starting vertex in a graph. At this scale, reproducibility becomes wishful thinking without adequate tooling and data management.

In Chapter 3 on page 17 we present the high-level architecture of the software pipeline we built to produce, aggregate, and analyse the results from all our experiments. The version of the code used in this thesis is available on Zenodo [89], the current version of the code can be found at `https://github.com/merijn/Belewitte`.

The core component of our pipeline is an SQLite-based database. This database stores all data related to our experiments and their configuration. This includes information related to possible experiment configurations, such as:

- known hardware platforms,
- known algorithms,
- known implementations per algorithm,
- the commit of each of the known implementations,
- known datasets, and
- known graphs per dataset.

The database also stores all the configurations for which experiments need to be run and/or have run. One of our tools examines which configurations have experiments that have not run yet, and runs any of the necessary benchmarks. The results of these experiments are added to the database, these include:

- timings of various stages of the algorithm,
- structural properties of the graph,
- properties of the algorithm that vary at run time,
- the date and time of the benchmark,
- whether the result of the experiment was validated[1],

All of our tools for analysis and plotting also interact with this database. The database stores the configuration used for, for example, our Binary Decision Tree (BDT) work, allowing us to (re)evaluate a model against both the exact same data used for training and comparing it against other data selections. The trained BDT models of our work, as well as the important metadata from the training process are stored within the database too.

There are additional benefits to packing all this data and metadata into a single SQLite database, besides the reproducibility aspect. Having all the data about experiment configuration, results, and analysis configuration in a single database makes tracking the provenance of results much simpler.

Furthermore, the single file format and wide support for SQLite make it trivial to share entire result sets with other researchers, letting them build their own work on top of the existing results without having to redo all the time consuming benchmarks themselves. The interested reader can find all the data that went into this thesis archived on Zenodo [90].

Software engineering and programming work is often viewed as distinct from "real computer science" and rarely as a scientific contribution on its own. In our opinion this software pipeline is an integral part of the scientific contributions in this thesis. It forms the backbone of the methodology we use in this thesis and allows other to do similar investigations on other graph algorithms, datasets, hardware, and/or implementations.

9.1.1.2 Quantifying the Performance Impact

The impact of input data — graphs, in our case — on irregular algorithm is well-known in HPC and GPGPU communities. This irregularity was already highlighted as the main challenge in high-performance graph processing in 2007 [61].

[1] i.e., whether it was the same as all other implementations.

Since then there has been little to no systematic investigation into the link between graph structure and the performance of different implementation techniques. This impact is generally assumed to be significant, but there is no work in quantifying what the impact actually is.

In Chapter 4 on page 41 we start by investigating the behaviour of neighbour iteration. Neighbour iteration appears as a primitive graph operation in many algorithms, making these results applicable beyond the PageRank and BFS algorithms we use. In Section 4.2 on page 43 we discuss the possible parallelisation strategies for neighbour iteration on GPGPU.

In Sections 4.4 and 4.5 on page 50 and on page 58 we show that there is, indeed, a significant variation in the performance of each parallelisation strategy across different input graphs. For some graphs the difference between the best and worst parallelisation strategy can be several orders of magnitude.

Furthermore, we show that the performance of BFS does not just vary with the input graph, but also with the stage of the BFS traversal. We show that correctly predicting the best implementation for each BFS step can produce significant performance gains.

9.1.1.3 Relation Between Graph Structure and Parallelisation

When we try to relate graph structure to the behaviour of different parallelisation strategies, we immediately run into a key problem: What is the structure of a graph?

Social network graphs, road networks, collaboration networks, and various other graphs are "obviously" different to the human eye, but classifying these differences is hard. Additionally, human intuition breaks down when comparing graphs with thousands or millions of vertices and edges.

We can attempt to classify the structure of a graph based in properties of the graph. However, people keep inventing new metrics and the existing ones are highly correlated. For example, increasing the edge count of a graph will, invariably increase the triangle count and lower the diameter and mean distance between nodes.

There is no consensus on which properties adequately classify the structure of a graph. So we are left to experimentally determine which properties are most important. Our intuition led us to believe that, in addition to the size of the graph, the degree distribution is probably one of the most important factors.

So in this thesis we have limited ourselves to: vertex count, edge count, and a number of properties related to degree distribution. All of these can be computed relatively cheaply even for very big graphs, as the complexity of computing them is $O(|V|)$.

Systematic Graph Generation

Our first attempt to relate the structural properties of graphs to the performance of parallelisation strategies was based on systematic benchmarking. The idea was to generate a dataset of synthetic graphs with a uniform distribution of values across each property of interest.

This would allow us to benchmark our parallelisation strategies against this systematic dataset and isolate the impact of each parameter on the performance of each parallelisation strategy.

None of the existing graph datasets met the requirements for this approach and none of the existing synthetic graph generators was able to generate the graphs we needed.

We set out to build our own synthetic graph generator. In Chapter 5 on page 67 we present our evolutionary computing based graph generator and the design of the evolutionary computing approach it uses.

While our graph generator was successful at generating graphs at small scales, it failed to scale up to the larger graph sizes we need to draw any conclusions about the link between graph structure and parallelisation.

Analytical Model

We were forced to abandon the systematic benchmarking approach, after our graph generator did not work out. Our next step was to try to tackle the problem using analytical methods.

In Chapter 6 on page 85 we presented workload models for each of our PageRank parallelisation strategies. We then validated our workload models against the behaviour observed by NVIDIA's profiling tools. There was a good correspondence between the workload models and the results observed using the profiling tools.

Our workload models are based on the memory accesses performed by each implementation. They only model the total amount of memory accesses and do not account for the parallelisation of these accesses. We showed that the edge-centric parallelisation strategy *always* performs more accesses than the other strategies, yet it is one of the fastest overall implementations.

We conclude that, as expected, the total workload is *not* sufficient to predict the performance of our implementations. Our experiments with different graph orderings show that it is not possible to statically approximate the parallel execution costs of our workload models. We conclude that accurately predicting the performance of the different parallelisation strategies requires modelling the dynamic behaviour of the GPU.

We do not believe that modelling the parallel execution is inherently impossible. However, details about the internals of GPUs are limited. Without detailed information from manufacturers, these internals have to

be reverse engineered from observation. We consider this an infeasible amount of work, especially for a single thesis.

BDT Modelling

Our final effort to relate graph structure to parallelisation strategy was via machine learning on our real world datasets. In Chapter 7 on page 99 we used the 247 graphs from the KONECT [51] dataset, and ran our implementations against all of them. We show that a BDT model trained on our results for the KONECT dataset can predict the best parallelisation strategy for a graph from its structural properties.

In Section 8.1 on page 112 we show that our BDT models are not simply memorising the training data. We vary the size of the training set used to train our models and show that models trained on a fraction of our result set are still effective.

In Section 8.2 on page 115 we expand on this by showing that our BDT models are not limited to the dataset they were trained on. Showing that they capture (part of) the link between graph structure and parallelisation strategy, rather than memorising results of a specific result set.

9.1.1.4 Exploiting the Relation Between Graph Structure and Parallelisation

Above we established that BDTs give us a way to predict the best performing parallelisation strategy from a graph's structure. In Section 7.3 on page 104 we show that our dynamically switching Mix-and-Match implementation is 1.6× faster than the (non-existent) non-switching oracle and 3.4× faster than the overall fastest implementation.

Having established that this relationship *can* be exploited, the next step, in Chapter 8 on page 111, is to investigate how general this relationship is and how portable our BDT models are. In Section 8.2 on page 115 we showed that the relationship between graph structure and parallelisation strategy performance is preserved across datasets, if the training dataset is sufficiently general. A definition of "sufficiently general" is left for future work.

9.1.1.5 GPU Invariance

One of our starting assumptions was that the biggest performance impact of GPGPU programming comes from mapping graph algorithms to their Single Instruction, Multiple Threads (SIMT) programming model. Leading us to belief that we can safely ignore the specifics of different GPU generations and assume that the relative performance of different parallelisation strategies stays fairly constant across GPU generations.

This is a pretty big assumption, so in Section 8.3 on page 119 we showed, experimentally, that this assumption is true enough to be useful. To do this we repeated the same experiments on 4 separate GPUs from 2012 through 2018, using 3 different microarchitectures. We trained BDT models on the result set of each of these GPUs; then we evaluated the effectiveness of each model against the results from the other GPUs.

If the GPU hardware had no impact we would expect almost no difference between the 4 sets of models or between the 4 evaluation sets. Of course, this is not the case, as there is some variance. However, this variance is not sufficient to completely invalidate our assumption. We note that our models outperform the non-model implementations, regardless which dataset the models were trained on.

From this we conclude that the most significant link between graph structure and parallelisation strategy follows from the SIMT programming model and that variations in GPU hardware only play a limited role, allowing us to generalise findings across multiple GPUs or GPU generations to some extent.

9.1.2 Methodology

Graph processing and performance engineering are both incredibly complicated subjects. The deeper you dive into either topic, the more new problems, complications, and degrees of freedom you discover. There are more unknowns than can be addressed in a lifetime, let alone a thesis, even when we only consider a niche like performance engineering graph processing on GPGPU.

Our goal in this thesis was never to provide a comprehensive solution to the problems of performance engineering GPU graph processing[2]. Instead, we hope to provide a methodology and starting point for others to explore this problem area further.

Our software pipeline from Chapter 3 on page 17 [89] is an essential part of this work. The results from Chapters 7 and 8 would not have been possible without it. Furthermore, it provides a straightforward and automated way for other researchers to explore and experiment and continue this investigation with other graph datasets, other hardware, or other algorithms.

Repeating out experiments with different datasets and hardware can be done without any programming effort. Other graph algorithms will require adding these algorithms to our kernel runner (or changing the output of existing software to match the expected output of our runner).

The infrastructure for running, aggregating, and analysing new experiments can be used unchanged with any algorithm that can be implemented

[2] Well…it was, but those hopes were squashed quite quickly!

as a Bulk Synchronous Parallel (BSP) algorithm with deterministic supersteps.

The SQLite database for storing results and experiments can also easily be extended to include other machine learning approaches or analyses besides the current BDT models. It is easy to write new tools that build on top of the results of other scientists, since SQLite is a widely supported format with libraries in almost all programming languages.

We think that this integrated and systematic approach is the key to successful and reproducible research in empirical computer science.

9.1.3 Contributions

The main contributions of this thesis are:

- A tool for gathering performance data of graph algorithms, running data analyses on these results, and evaluating models against our empirical data. [89]
  See Chapter 3 on page 17.

- A quantification of the performance impact of different parallelisation strategies on graph algorithm performance. [94]
  See Chapter 4 on page 41.

- A graph generator based on evolutionary computing to generate input graphs for our experimentation. [95]
  See Chapter 5 on page 67.

- A workload model for these parallelisation strategies and demonstrating the infeasibility of using this workload model for performance prediction in the context of GPU execution. [94]
  See Chapter 6 on page 85.

- A workflow for creating BDT models for predicting the performance of GPU graph algorithms. [92, 93]
  See Chapters 3 and 7 on page 17 and on page 99.

- A proof of concept graph traversal that uses the above BDT model to get better performance by switching between implementations for different steps of the traversal. [92, 93]
  See Chapter 7 on page 99.

- An analysis of the portability of our BDT models across datasets and GPU architectures. [90]
  See Chapter 8 on page 111.

9.2 Future Work

During the work described in this thesis[3], there were many interesting ideas, questions, and variations to explore that we simply did not have the time to go into. In this section we highlight some areas of interest for further investigation and future work.

One of the most interesting investigations would be to repeat our methodology and BDT modelling approaches with additional GPU algorithms and see how well the results hold for these. Candidate algorithms are: Single-Source Shortest Path (SSSP), Betweenness Centrality (BC), or connected components. Our modelling approach should work with any BSP algorithm.

In Chapter 8 on page 111 we showed that BDT models trained on very limited amount of data still perform quite well. However, this investigation was quite limited and focussed on showing that our BDT models were not simply memorising their input data.

It is an open question what a "representative dataset" looks like. We noted that models trained on KONECT performed well on SNAP graphs, but not vice versa. We argued that this was likely due to the KONECT dataset being more diverse. However, there is still a lot of redundancy in the form of similar graphs in the KONECT dataset.

An interesting avenue of investigation would be to see how many and which graphs could be dropped from out KONECT training set without affecting the model quality and the transferability to SNAP.

A similar thing could be used to construct a classification for graphs, for a given graph, see how many and which graphs can be dropped from the training set without worsening the model's performance. Any graphs that must be kept to keep the model's performance are evidently in the same class.

In Chapters 4 and 6 on page 41 and on page 85 we noted that there are performance differences depending on the exact in memory representation of graphs. We did not examine these effects in depth, as this did not seem to effect the relative performance of different parallelisation strategies, but it is interesting to see if and how the performance can be optimised by altering how the graph's structure is stored in memory.

[3] And even while already writing it

Appendices

APPENDIX A

Current Database Schema

This section serves as a reference of the current version of our data format, clarifying all the gory details of the schema shown in Fig. 3.3 on page 29. Explaining which columns exist in each table and the meaning of the data in that column. For a higher level overview of how our tools interact with this data format we refer to Section 3.3 on page 30.

A.1 GlobalVars

```
CREATE TABLE IF NOT EXISTS "GlobalVars"
( "name" VARCHAR NOT NULL
, "value" VARCHAR NOT NULL
, PRIMARY KEY ("name")
, CONSTRAINT "UniqGlobal" UNIQUE ("name")
);
```

The `GlobalVars` table, as the name implies, stores global variables controlling the operation of the `Ingest`, `Model`, and `Plot` executables. The table consists of `name`–`value` pairs, while both columns are listed as text, the `value` column can use SQLite's dynamic typing to store values of different types as needed. In the current schema version this table only stores the "run command". That is, the runner command used to start benchmarking runs on systems that do not use SLURM [47].

A.2 Platform

```
CREATE TABLE IF NOT EXISTS "Platform"
( "id" INTEGER PRIMARY KEY
, "name" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, "flags" VARCHAR NULL
, "available" INTEGER NOT NULL DEFAULT 1
, "isDefault" BOOLEAN NOT NULL DEFAULT 0
, CONSTRAINT "UniqPlatform" UNIQUE ("name")
);
```

The `Platform` table stores a list of hardware platforms (in this thesis only Graphical Processing Units (GPUs), but the applicability is broader than that). The `id` column provides a unique key to refer to each platform. The `name` column gives the textual name of the platform. This textual name is dual purpose, it is used as "human-readable" platform name by the command line interface and as argument to the SLURM [47] runner to allocate a machine with the right platform.

The `prettyName` column gives an optional longer form name to be used in tables, graphs, and other output. The `flags` column is an optional way to specify what flags to use when running a set of experiments. If the value in the `flags` column is present, it will override the use of the `name` column in telling the SLURM runner what machine(s) to allocate.

The `available` column specifies the number of available machines with this platform (e.g., GPU) are available, limiting how many parallel jobs the runner will attempt to spawn. The `isDefault` column can only be true for a single row in the table and specifies which platform should be used as the default for to run non-benchmarking tasks on. This is the platform that will be used to compute graph properties and validate results. Ideally, the default platform should be one of the faster platforms or the one with the most machines available, to maximise the speed with which these tasks are completed.

A.3 Dataset

```
CREATE TABLE IF NOT EXISTS "Dataset"
( "id" INTEGER PRIMARY KEY
, "name" VARCHAR NOT NULL
, CONSTRAINT "UniqDataset" UNIQUE ("name")
);
```

The `Dataset` table defines which datasets existed, which can then be used for configuring experimental runs and computing statistics over the

runs on a specific dataset. The table consists of a unique numerical `id` column and a `name` column for unique textual names for the datasets.

A.4 Graph

```
CREATE TABLE IF NOT EXISTS "Graph"
( "id" INTEGER PRIMARY KEY
, "name" VARCHAR NOT NULL
, "path" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, "datasetId" INTEGER NOT NULL REFERENCES "Dataset"
, "timestamp" TIMESTAMP NOT NULL
, CONSTRAINT "UniqGraph" UNIQUE ("path")
, CONSTRAINT "UniqGraphName" UNIQUE ("name","datasetId")
);
```

The `Graph` table defines all the graphs available for experiments. For every graph we store a unique numerical `id`, a short textual `name` for command line interaction, a longer `prettyName` used in tables, graphs, and other output. The `path` column gives the filepath the graph file. The `datasetId` is a foreign key to the `Dataset` table, defining which dataset the graph belongs to. The `timestamp` table stores when the graph's information was added to the database, this is lets us determine if a graph was included in old experiments or the training/validation of older models stored in the database.

A.5 Algorithm

```
CREATE TABLE IF NOT EXISTS "Algorithm"
( "id" INTEGER PRIMARY KEY
, "name" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, CONSTRAINT "UniqAlgorithm" UNIQUE ("name")
);
```

The `Algorithm` table defines the algorithms available for experiments. For each algorithm we store a unique numerical `id`, a unique short textual `name` for command line interaction, and a longer `prettyName` used in tables, graphs, and other output.

A.6 Implementation

```
CREATE TABLE IF NOT EXISTS "Implementation"
( "id" INTEGER PRIMARY KEY
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "name" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, "flags" VARCHAR NULL
, "type" VARCHAR NOT NULL
, CONSTRAINT "UniqImpl" UNIQUE ("algorithmId","name")
);
```

The `Implementation` table defines all the available implementations for each `Algorithm`. We have the standard unique numerical `id` column, unique short textual `name` column, and longer textual `prettyName` column used in tables, graphs, and other output. Every `Implementation` has a foreign key `algorithmId` linking it to its corresponding `Algorithm` row.

By default the `name` column is used to tell the runner which implementation to use, but the `flags` column can be used to override this behaviour. When the `flags` column is present, those flags are passed to the runner instead of the `name`.

The `type` column distinguishes between `Core` implementations which are unique implementations of the algorithm and `Derived` implementations which are implementation that use our Binary Decision Tree (BDT) models to switch between different `Core` implementations at runtime.

A.7 VariantConfig

```
CREATE TABLE IF NOT EXISTS "VariantConfig"
( "id" INTEGER PRIMARY KEY
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "name" VARCHAR NOT NULL
, "flags" VARCHAR NULL
, "isDefault" BOOLEAN NOT NULL
, "timestamp" TIMESTAMP NOT NULL
, CONSTRAINT "UniqVariantConfig" UNIQUE ("algorithmId","name")
);
```

The `VariantConfig` table specifies the different ways an algorithm can be run on the same input. Not every algorithm will have multiple ways to run it. For example, Breadth-First Search (BFS) can be run with different root nodes on the same input, whereas PageRank only has one way to run it.

For every variant configuration we store a unique numerical `id`, the `algorithmId` referencing the `Algorithm` it belongs to, and a short textual `name`. The `flags` column specifies the flags needed to run the algorithm in this configuration. The `isDefault` column specifies which variant configuration is the default to use for each algorithm. The `timestamp` specifies when the configuration was added to the database, this is lets us determine if a variant configuration was included in old experiments or the training/validation of older models stored in the database.

A.8 Variant

```
CREATE TABLE IF NOT EXISTS "Variant"
( "id" INTEGER PRIMARY KEY
, "graphId" INTEGER NOT NULL REFERENCES "Graph"
, "variantConfigId" INTEGER NOT NULL REFERENCES "VariantConfig"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "result" BLOB NULL
, "maxStepId" INTEGER NOT NULL
, "propsStored" BOOLEAN NOT NULL
, "retryCount" INTEGER NOT NULL
, CONSTRAINT "UniqVariant" UNIQUE ("graphId","variantConfigId")
, CONSTRAINT "ForeignVariantConfig"
  FOREIGN KEY("variantConfigId","algorithmId")
  REFERENCES "VariantConfig"("id","algorithmId")
);
```

The `Variant` table stores results and metadata for every `Graph` & `VariantConfig` combination. A variant has a unique numerical `id`, and `graphId` & `variantConfgifId` foreign key references to the corresponding `Graph` and `VariantConfig`. The `algorithmId` column references the `Algorithm`, we use a foreign key constraint to ensure it corresponds to the `VariantConfig`'s `Algorithm`. This is redundant, but lets us reduce the number of joins in our Structured Query Language (SQL) queries, speeding them up.

For every `Variant` we store the `result`, which is a hash of the algorithm's output, used to check that all our implementation produce the same results. We also store the `maxStepId` which is the number of iterations/steps of our implementation it takes for computation to finish. The `propsStored` column tracks whether we already computed and stored the algorithm specific graph properties for this variant. The `retryCount` columns tracks the number of failed runs have been attempted for this variant, allowing us to stop scheduling experiments if they fail too many times.

A.9 RunConfig

```
CREATE TABLE IF NOT EXISTS "RunConfig"
( "id" INTEGER PRIMARY KEY
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "platformId" INTEGER NOT NULL REFERENCES "Platform"
, "datasetId" INTEGER NOT NULL REFERENCES "Dataset"
, "algorithmVersion" VARCHAR NOT NULL
, "repeats" INTEGER NOT NULL
, CONSTRAINT "UniqRunConfig"
  UNIQUE ("algorithmId","platformId","datasetId","algorithmVersion")
);
```

The `RunConfig` table stores sets of past and future benchmarking experiments. Each set has a unique numerical `id` and is made up of an algorithm to run, the platform to run it on, and the dataset whose graphs should be used as input. These are stored using the `algorithmId`, `platformId`, and `datasetId` foreign references, respectively. The `algorithmVersion` stores the version of the implementations that was/will be used for the experiments. The `repeats` column stores the number of runs the results of each implementation should be averaged over.

A.10 Run

```
CREATE TABLE IF NOT EXISTS "Run"
( "id" INTEGER PRIMARY KEY
, "runConfigId" INTEGER NOT NULL REFERENCES "RunConfig"
, "variantId" INTEGER NOT NULL REFERENCES "Variant"
, "implId" INTEGER NOT NULL REFERENCES "Implementation"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "timestamp" TIMESTAMP NOT NULL
, "validated" BOOLEAN NOT NULL
, CONSTRAINT "UniqRun"
  UNIQUE ("runConfigId","variantId","implId","algorithmId")
, CONSTRAINT "ForeignRunConfig"
  FOREIGN KEY("runConfigId","algorithmId")
  REFERENCES "RunConfig"("id","algorithmId")
, CONSTRAINT "ForeignVariant"
  FOREIGN KEY("variantId","algorithmId")
  REFERENCES "Variant"("id","algorithmId")
, CONSTRAINT "ForeignImplementation"
  FOREIGN KEY("implId","algorithmId")
  REFERENCES "Implementation"("id","algorithmId")
);
```

The `Run` table stores information tracking which of the runs specified by the `RunConfig` table have been completed already. A `Run` has a unique numerical `id`. `Run`s correspond to a combination of a specific `RunConfig`, `Implementation` of the algorithm, and `Variant` (e.g., combination of graph and algorithm specific configuration). These are stored using the `runConfigId`, `variantId`, and `implId` foreign references, respectively.

The `algorithmId` foreign reference is redundant and only present to enforce consistency and simplify certain SQL queries. The `timestamp` column stores the Coordinated Universal Time (UTC) time when the `Run` completed and was stored. This lets us accurately reconstruct historical training/validation sets used when training models. The `validated` column tracks whether the `Run`'s output data match the reference results for the given `Variant`, acting as verification that all implementations produce identical results.

A.11 PropertyName

```
CREATE TABLE IF NOT EXISTS "PropertyName"
( "id" INTEGER PRIMARY KEY
, "property" VARCHAR NOT NULL
, "isStepProp" BOOLEAN NOT NULL
, CONSTRAINT "UniqProperty" UNIQUE ("property")
);
```

The `PropertyName` table stores the properties tracked by the framework. Each property has a unique numerical `id`, the `property` column stores a unique short textual name, and the `isStepProp` column, as the name implies, stores whether the property is of an algorithms (super)steps or a property of a graph.

A.12 GraphPropValue

```
CREATE TABLE IF NOT EXISTS "GraphPropValue"
( "graphId" INTEGER NOT NULL REFERENCES "Graph"
, "propId" INTEGER NOT NULL REFERENCES "PropertyName"
, "value" REAL NOT NULL
, PRIMARY KEY ("graphId","propId")
, CONSTRAINT "UniqGraphPropValue" UNIQUE ("graphId","propId")
);
```

The `GraphPropValue` table stores the values of graph properties. Each row is identified by a unique pair of a graph and property, identified by the

`graphId` and `propId` foreign references, respectively. The `value` column stores the value of the associated graph and property combination.

A.13 StepProp

```
CREATE TABLE IF NOT EXISTS "StepProp"
( "propId" INTEGER NOT NULL REFERENCES "PropertyName"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, PRIMARY KEY ("propId","algorithmId")
, CONSTRAINT "UniqStepProp" UNIQUE ("propId","algorithmId")
);
```

The `StepProp` table allows multiple algorithms to share step properties with the same name. Each row consists of a combination of a `propId` and `algorithmId` foreign references, indicating that the property exists for the given algorithm.

A.14 StepPropValue

```
CREATE TABLE IF NOT EXISTS "StepPropValue"
( "variantId" INTEGER NOT NULL REFERENCES "Variant"
, "stepId" INTEGER NOT NULL
, "propId" INTEGER NOT NULL REFERENCES "PropertyName"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "value" REAL NOT NULL
, PRIMARY KEY ("variantId","stepId","propId")
, CONSTRAINT "UniqStepPropValue"
  UNIQUE ("variantId","stepId","propId")
, CONSTRAINT "ForeignStepProp"
  FOREIGN KEY("algorithmId","propId")
  REFERENCES "StepProp"("algorithmId","propId")
, CONSTRAINT "ForeignVariant"
  FOREIGN KEY("variantId","algorithmId")
  REFERENCES "Variant"("id","algorithmId")
);
```

The `StepPropValue` table stores the values of step properties. Step properties are runtime values of a given algorithm, as opposed to the static properties of input graphs, as such they can differ between variants of the same graph. Consider the case of BFS we would not expect the same number of steps nor same values for BFS traversals starting from different roots.

Step property values are therefore associated with a specific variant and step. The `propId` references a step property and is associated to a

variant via the `variantId` foreign reference and the step number `stepId`. The `value` column stores the property's value. The `algorithmId` foreign reference is redundant (inferrible from the variant), but stored for query optimisation purposes.

A.15 TotalTimer

```
CREATE TABLE IF NOT EXISTS "TotalTimer"
( "runId" INTEGER NOT NULL REFERENCES "Run"
, "name" VARCHAR NOT NULL
, "minTime" REAL NOT NULL
, "avgTime" REAL NOT NULL
, "maxTime" REAL NOT NULL
, "stdDev" REAL NOT NULL
, PRIMARY KEY ("runId","name")
);
```

The `TotalTimer` stores named overall timers for an associated `Run`. A timer consists of a `runId` foreign reference, a unique (per `Run`) short textual name, and minimum (`minTime`), average (`avgTime`), maximum (`maxTime`), and standard deviations (`stdDev`). The number of samples these timing aggregates are computed from are determined by the number of repeats specified in the corresponding `RunConfig`.

A.16 StepTimer

```
CREATE TABLE IF NOT EXISTS "StepTimer"
( "runId" INTEGER NOT NULL REFERENCES "Run"
, "variantId" INTEGER NOT NULL REFERENCES "Variant"
, "stepId" INTEGER NOT NULL
, "name" VARCHAR NOT NULL
, "minTime" REAL NOT NULL
, "avgTime" REAL NOT NULL
, "maxTime" REAL NOT NULL
, "stdDev" REAL NOT NULL
, PRIMARY KEY ("runId","stepId","name")
, CONSTRAINT "ForeignRun"
  FOREIGN KEY("runId","variantId")
  REFERENCES "Run"("id","variantId")
);
```

The `StepTimer` stores, as implied by the name, the named step timers for an associated `Run`. The `runId`, `name`, `minTime`, `avgTime`, `maxTime`, `stdDev` fields are identical to their `TotalTimer` equivalents.

In addition, we also store the numerical `stepId` and a `variantId` foreign reference to the corresponding variant. The `variantId` is redundant but stored for input validation and query optimisation purposes.

A.17 ExternalImpl

```
CREATE TABLE IF NOT EXISTS "ExternalImpl"
( "id" INTEGER PRIMARY KEY
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "name" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, CONSTRAINT "UniqExternalImpl" UNIQUE ("algorithmId","name")
);
```

The `ExternalImpl` table stores "external" implementations, that is, names of implementations not included in nor runnable by our tool, but whose results are used for comparison tables and plots. They consist of a unique numerical `id`, a `algorithmId` foreign reference to the corresponding algorithm, a short textual `name` for command line interaction, and a longer `prettyName` used for tables, graphs, and other output.

A.18 ExternalTimer

```
CREATE TABLE IF NOT EXISTS "ExternalTimer"
( "platformId" INTEGER NOT NULL REFERENCES "Platform"
, "variantId" INTEGER NOT NULL REFERENCES "Variant"
, "implId" INTEGER NOT NULL REFERENCES "ExternalImpl"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "name" VARCHAR NOT NULL
, "minTime" REAL NOT NULL
, "avgTime" REAL NOT NULL
, "maxTime" REAL NOT NULL
, "stdDev" REAL NOT NULL
, PRIMARY KEY ("platformId","variantId","implId","algorithmId","name")
, CONSTRAINT "ForeignVariant"
  FOREIGN KEY("variantId","algorithmId")
  REFERENCES "Variant"("id","algorithmId")
, CONSTRAINT "ForeignExternalImpl"
  FOREIGN KEY("implId","algorithmId")
  REFERENCES "ExternalImpl"("id","algorithmId")
);
```

The `ExternalTimer` table stores the results of benchmarks using external implementations. These timings correspond to a combination of a

platform, variant, and external implementation, references by `platformId`, `variantId`, and `implId`. The `algorithmId` is redundant and inferable, but included for data validation and query optimisation purposes.

The `name` column corresponds to a timer name, in case of multiple timers per implementation. The `minTime`, `avgTime`, `maxTime`, and `stdDev` correspond to the minimum, average, maximum times, and standard deviation, respectively.

A.19 PredictionModel

```
CREATE TABLE IF NOT EXISTS "PredictionModel"
( "id" INTEGER PRIMARY KEY
, "platformId" INTEGER NOT NULL REFERENCES "Platform"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "algorithmVersion" VARCHAR NOT NULL
, "name" VARCHAR NOT NULL
, "prettyName" VARCHAR NULL
, "description" VARCHAR NULL
, "model" BLOB NOT NULL
, "skipIncomplete" BOOLEAN NOT NULL
, "legacyTrainFraction" REAL NOT NULL
, "trainGraphs" REAL NOT NULL
, "trainVariants" REAL NOT NULL
, "trainSteps" REAL NOT NULL
, "trainSeed" INTEGER NOT NULL
, "totalUnknownCount" INTEGER NOT NULL
, "timestamp" TIMESTAMP NOT NULL
, CONSTRAINT "UniqModel" UNIQUE ("name")
);
```

The `PredictionModel` table stores trained models and metadata about the training process. Each model has a unique numerical `id`, a short textual `name`, and was trained on data from a specific platform (`platformId`) and algorithm (`algorithmId`). Each model has an optional `description` describing it and an optional `prettyName` for use in output. The number of leaf nodes with an unknown prediction is stored in the `totalUnknownCount` column.

The `model` itself is stored as a binary blob which can be serialised into C++ code by our tools. The `timestamp` indicates when the model was trained, allowing us to reconstruct which results and measurements were available at the time. This guarantees that graphs and tables are reproducible even when extra results are added to the database later.

The training sets and validation sets are computed via a deterministic Pseudo Random Number Generator (PRNG)-based sampling algorithm. The metadata controlling this sampling are:

legacyTrainFraction,
: whether to use the legacy, not fully deterministic, sampling strategy. Exists for backwards compatibility with very old initial experiments.

skipIncomplete,
: only include results for graphs here *all* implementations have validated measurements available.

trainGraphs,
: the fraction of all graphs that should be included in the training data.

trainVariants,
: the fraction of all variants that should be included in the training data.

trainSteps,
: the fraction of all steps that should be included in the training data.

trainSeed,
: the seed for the sampling algorithm.

A.20 ModelTrainDataset

```
CREATE TABLE IF NOT EXISTS "ModelTrainDataset"
( "modelId" INTEGER NOT NULL REFERENCES "PredictionModel"
, "datasetId" INTEGER NOT NULL REFERENCES "Dataset"
, PRIMARY KEY ("modelId","datasetId")
);
```

Models can be trained on data from graphs from multiple datasets. The `ModelTrainDataset` stores which datasets a `PredictionModel` was trained on, by linking the to one or more datasets via the `modelId` and `datasetId` foreign references.

A.21 ModelProperty

```
CREATE TABLE IF NOT EXISTS "ModelProperty"
( "modelId" INTEGER NOT NULL REFERENCES "PredictionModel"
, "propId" INTEGER NOT NULL REFERENCES "PropertyName"
, "propertyIdx" INTEGER NOT NULL
, "importance" REAL NOT NULL
, PRIMARY KEY ("modelId","propId")
);
```

The `ModelProperty` tables stores the properties that were used for the training of a model. The `modelId` and `propId` foreign references links models and properties. The BDT models expect an array or property values to perform a prediction. The numerical `propertyIdx` specifies which index the model expects the property's values to be in. Finally, the `importance` column store the Gini importance of the property. That is, a measure indicating how much "predictive power" that property contributes to the model. Allowing us to exam the properties that have the largest impact on a given model.

A.22 UnknownPrediction

```
CREATE TABLE IF NOT EXISTS "UnknownPrediction"
( "id" INTEGER PRIMARY KEY
, "modelId" INTEGER NOT NULL REFERENCES "PredictionModel"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, "unknownSetId" INTEGER NOT NULL
, "count" INTEGER NOT NULL
, CONSTRAINT "UniqUnknownPrediction" UNIQUE ("modelId","unknownSetId")
, CONSTRAINT "ForeignPredictionModel"
  FOREIGN KEY("modelId","algorithmId")
  REFERENCES "PredictionModel"("id","algorithmId")
);
```

As described in Section 7.1 on page 100, it is possible for BDT models to have "undecided" leaves. That is, leaf nodes in the tree where there is no single winning prediction. The `UnknownPrediction` table documents all unique sets of conflicting predictions in a model's leaf nodes. We assign these sets a unique numerical `id`, relate them to their corresponding model via the `modelId` foreign reference. The numerical `unknownSetId` defines the numerical id used within the BDT model to identify leaves corresponding to this set. The `count` column documents how frequently this specific set occurs within the BDT model. Finally, the `algorithmId` foreign reference is redundant, but include for data validation purposes.

A.23 UnknownPredictionSet

```
CREATE TABLE IF NOT EXISTS "UnknownPredictionSet"
( "unknownPredId" INTEGER NOT NULL REFERENCES "UnknownPrediction"
, "implId" INTEGER NOT NULL REFERENCES "Implementation"
, "algorithmId" INTEGER NOT NULL REFERENCES "Algorithm"
, PRIMARY KEY ("unknownPredId","implId")
, CONSTRAINT "ForeignUnknownPrediction"
  FOREIGN KEY("unknownPredId","algorithmId")
  REFERENCES "UnknownPrediction"("id","algorithmId")
);
```

The `UnknownPredictionSet` stores which of the implementations are included within an `UnknownPrediction` in the BDT. Rows correspond to a specific `UnknownPrediction` via the `unknownPredId` foreign reference and to an implementation via the `implId` foreign reference. The `algorithmId` is used for data validation and sanity checking.

APPENDIX B

PageRank PTX Code

Parallel Thread Execution (PTX) assembly output for PageRank Compute Unified Device Architecture (CUDA) kernels with symbol names demangled.

B.1 Edge List

```
;
; Generated by NVIDIA NVVM Compiler
;
; Compiler Build ID: CL-24817639
; Cuda compilation tools, release 10.0, V10.0.130
; Based on LLVM 3.4svn

.version 6.3
.target sm_53
.address_size 64

.visible .entry edgeListPageRank(
    .param .u64 edgeListPageRank_graph,
    .param .u64 edgeListPageRank_degrees,
    .param .u64 edgeListPageRank_pagerank,
    .param .u64 edgeListPageRank_new_pagerank
)
{
    .reg .pred      %p<4>;
    .reg .f32       %f<8>;
```

```
    .reg .b32       %r<10>;
    .reg .b64       %rd<30>;

    ld.param.u64    %rd13, [edgeListPageRank_graph];
    ld.param.u64    %rd10, [edgeListPageRank_degrees];
    ld.param.u64    %rd11, [edgeListPageRank_pagerank];
    ld.param.u64    %rd12, [edgeListPageRank_new_pagerank];

;pagerank/edgelist.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    mov.u32         %r1, %ntid.x;
    mov.u32         %r4, %ctaid.x;
    mov.u32         %r5, %tid.x;
    mad.lo.s32      %r6, %r1, %r4, %r5;
    cvt.u64.u32     %rd29, %r6;
    cvta.to.global.u64      %rd14, %rd13;

;pagerank/edgelist.cu:12:
;uint64_t size = graph->edge_count;
    .loc 1 12 19
    add.s64         %rd2, %rd14, 8;
    ld.global.u64   %rd3, [%rd14+8];

;pagerank/edgelist.cu:14:
;for ( uint64_t idx = startIdx
     ; idx < size
     ; idx += blockDim.x * gridDim.x) {
    .loc 1 14 5
    setp.ge.u64     %p1, %rd29, %rd3;
    @%p1 bra        BB0_5;

;pagerank/edgelist.cu:14:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 14 47
    mov.u32         %r7, %nctaid.x;
    mul.lo.s32      %r8, %r7, %r1;
    cvt.u64.u32     %rd4, %r8;

;pagerank/edgelist.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
```

```
    cvta.to.global.u64      %rd5, %rd11;
    cvta.to.global.u64      %rd6, %rd12;
    cvta.to.global.u64      %rd22, %rd10;

BB0_2:
;pagerank/edgelist.cu:15:
;uint64_t origin = graph->inEdges[idx];
    .loc 1 15 25
    ld.global.u64   %rd15, [%rd2+8];
    cvta.to.global.u64      %rd16, %rd15;
    shl.b64         %rd17, %rd29, 2;
    add.s64         %rd18, %rd16, %rd17;
    ld.global.u32   %r9, [%rd18];
    cvt.u64.u32     %rd8, %r9;

;pagerank/edgelist.cu:16:
;uint64_t destination = graph->outEdges[idx];
    .loc 1 16 30
    ld.global.u64   %rd19, [%rd2+16];
    cvta.to.global.u64      %rd20, %rd19;
    add.s64         %rd21, %rd20, %rd17;
    ld.global.u32   %r2, [%rd21];

;pagerank/edgelist.cu:18:
;unsigned degree = degrees[origin];
    .loc 1 18 25
    mul.wide.u32    %rd23, %r9, 4;
    add.s64         %rd24, %rd22, %rd23;
    ld.global.u32   %r3, [%rd24];

;pagerank/edgelist.cu:20:
;if (degree != 0) new_rank = pagerank[origin] / degree;
    .loc 1 20 9
    setp.eq.s32     %p2, %r3, 0;
    mov.f32         %f7, 0f00000000;

;pagerank/edgelist.cu:20:
;if (degree != 0) new_rank = pagerank[origin] / degree;
    .loc 1 20 9
    @%p2 bra        BB0_4;

;pagerank/edgelist.cu:20:
;if (degree != 0) new_rank = pagerank[origin] / degree;
    .loc 1 20 26
```

```
    shl.b64         %rd25, %rd8, 2;
    add.s64         %rd26, %rd5, %rd25;
    cvt.rn.f32.u32  %f4, %r3;
    ld.global.f32   %f5, [%rd26];
    div.rn.f32      %f7, %f5, %f4;

BB0_4:
;pagerank/edgelist.cu:21:
;atomicAdd(&new_pagerank[destination], new_rank);
    .loc 1 21 9
    mul.wide.u32    %rd27, %r2, 4;
    add.s64         %rd28, %rd6, %rd27;

    .loc 3 77 10
    atom.global.add.f32     %f6, [%rd28], %f7;

;pagerank/edgelist.cu:14:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 14 47
    add.s64         %rd29, %rd4, %rd29;

;pagerank/edgelist.cu:14:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 14 5
    setp.lt.u64     %p3, %rd29, %rd3;
    @%p3 bra        BB0_2;

BB0_5:
;pagerank/edgelist.cu:23:
;}
    .loc 1 23 1
    ret;
}
```

B.2 Vertex Push

```
;
; Generated by NVIDIA NVVM Compiler
;
```

```
; Compiler Build ID: CL-24817639
; Cuda compilation tools, release 10.0, V10.0.130
; Based on LLVM 3.4svn

.version 6.3
.target sm_53
.address_size 64

.visible .entry vertexPushPageRank(
    .param .u64 vertexPushPageRank_graph,
    .param .u64 vertexPushPageRank_unused,
    .param .u64 vertexPushPageRank_pagerank,
    .param .u64 vertexPushPageRank_new_pagerank
)
{
    .reg .pred      %p<10>;
    .reg .f32       %f<16>;
    .reg .b32       %r<36>;
    .reg .b64       %rd<53>;

    ld.param.u64    %rd7, [vertexPushPageRank_graph];
    ld.param.u64    %rd8, [vertexPushPageRank_pagerank];
    ld.param.u64    %rd9, [vertexPushPageRank_new_pagerank];

;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd10, %rd7;
    mov.u32         %r12, %ntid.x;
    mov.u32         %r13, %ctaid.x;
    mov.u32         %r14, %tid.x;
    mad.lo.s32      %r15, %r12, %r13, %r14;
    cvt.u64.u32     %rd52, %r15;

;pagerank/vertex_push.cu:12:
;uint64_t size = graph->vertex_count;
    .loc 1 12 19
    ld.global.u64   %rd2, [%rd10];

;pagerank/vertex_push.cu:17:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 17 5
```

```
    setp.ge.u64     %p1, %rd52, %rd2;
    @%p1 bra        BB0_14;

    mov.f32         %f15, 0f00000000;

;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd17, %rd8;

BB0_2:
;pagerank/vertex_push.cu:18:
;unsigned *vertices = graph->vertices;
    .loc 1 18 28
    ld.global.u64   %rd12, [%rd10+16];

;pagerank/vertex_push.cu:19:
;unsigned *edges = graph->edges;
    .loc 1 19 25
    cvta.to.global.u64      %rd13, %rd12;
    ld.global.u64   %rd14, [%rd10+24];

;pagerank/vertex_push.cu:20:
;unsigned start = vertices[idx];
    .loc 1 20 24
    cvta.to.global.u64      %rd4, %rd14;
    shl.b64         %rd15, %rd52, 2;
    add.s64         %rd16, %rd13, %rd15;

;pagerank/vertex_push.cu:21:
;unsigned end = vertices[idx + 1];
    .loc 1 21 22
    ld.global.u32   %r1, [%rd16+4];

;pagerank/vertex_push.cu:20:
;unsigned start = vertices[idx];
    .loc 1 20 24
    ld.global.u32   %r2, [%rd16];

;pagerank/vertex_push.cu:23:
;degree = end - start;
    .loc 1 23 9
    sub.s32         %r3, %r1, %r2;
```

```
;pagerank/vertex_push.cu:25:
;if (degree != 0) outgoingRank = pagerank[idx] / degree;
    .loc 1 25 9
    setp.eq.s32     %p2, %r1, %r2;
    @%p2 bra        BB0_4;

;pagerank/vertex_push.cu:25:
;if (degree != 0) outgoingRank = pagerank[idx] / degree;
    .loc 1 25 26
    add.s64         %rd19, %rd17, %rd15;
    cvt.rn.f32.u32  %f5, %r3;
    ld.global.f32   %f6, [%rd19];
    div.rn.f32      %f15, %f6, %f5;

BB0_4:
;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 9
    setp.le.u32     %p3, %r1, %r2;
    @%p3 bra        BB0_13;

    and.b32         %r16, %r3, 3;
    setp.eq.s32     %p4, %r16, 0;
    @%p4 bra        BB0_11;

    setp.eq.s32     %p5, %r16, 1;
    @%p5 bra        BB0_10;

    setp.eq.s32     %p6, %r16, 2;
    @%p6 bra        BB0_9;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd20, %r2, 4;
    add.s64         %rd21, %rd4, %rd20;
    ld.global.u32   %r19, [%rd21];

;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd22, %rd9;

;pagerank/vertex_push.cu:28:
```

```
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd23, %r19, 4;
    add.s64         %rd24, %rd22, %rd23;

    .loc 3 77 10
    atom.global.add.f32     %f7, [%rd24], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r2, %r2, 1;

BB0_9:
;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd25, %r2, 4;
    add.s64         %rd26, %rd4, %rd25;
    ld.global.u32   %r20, [%rd26];

;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd27, %rd9;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd28, %r20, 4;
    add.s64         %rd29, %rd27, %rd28;

    .loc 3 77 10
    atom.global.add.f32     %f8, [%rd29], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r2, %r2, 1;

BB0_10:
;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
```

```
    mul.wide.u32    %rd30, %r2, 4;
    add.s64         %rd31, %rd4, %rd30;
    ld.global.u32   %r21, [%rd31];

;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd32, %rd9;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd33, %r21, 4;
    add.s64         %rd34, %rd32, %rd33;

    .loc 3 77 10
    atom.global.add.f32     %f9, [%rd34], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r2, %r2, 1;

BB0_11:
;pagerank/vertex_push.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd5, %rd9;
    setp.lt.u32     %p7, %r3, 4;
    @%p7 bra        BB0_13;

BB0_12:
;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd35, %r2, 4;
    add.s64         %rd36, %rd4, %rd35;
    ld.global.u32   %r22, [%rd36];
    mul.wide.u32    %rd37, %r22, 4;
    add.s64         %rd38, %rd5, %rd37;

    .loc 3 77 10
    atom.global.add.f32     %f10, [%rd38], %f15;
```

```
;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r23, %r2, 1;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd39, %r23, 4;
    add.s64         %rd40, %rd4, %rd39;
    ld.global.u32   %r24, [%rd40];
    mul.wide.u32    %rd41, %r24, 4;
    add.s64         %rd42, %rd5, %rd41;

    .loc 3 77 10
    atom.global.add.f32     %f11, [%rd42], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r25, %r2, 2;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd43, %r25, 4;
    add.s64         %rd44, %rd4, %rd43;
    ld.global.u32   %r26, [%rd44];
    mul.wide.u32    %rd45, %r26, 4;
    add.s64         %rd46, %rd5, %rd45;

    .loc 3 77 10
    atom.global.add.f32     %f12, [%rd46], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r27, %r2, 3;

;pagerank/vertex_push.cu:28:
;atomicAdd(&new_pagerank[edges[i]], outgoingRank);
    .loc 1 28 13
    mul.wide.u32    %rd47, %r27, 4;
    add.s64         %rd48, %rd4, %rd47;
```

```
    ld.global.u32   %r28, [%rd48];
    mul.wide.u32    %rd49, %r28, 4;
    add.s64         %rd50, %rd5, %rd49;

    .loc 3 77 10
    atom.global.add.f32     %f13, [%rd50], %f15;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 43
    add.s32         %r2, %r2, 4;

;pagerank/vertex_push.cu:27:
;for (unsigned i = start; i < end; i++) {
    .loc 1 27 9
    setp.lt.u32     %p8, %r2, %r1;
    @%p8 bra        BB0_12;

BB0_13:
;pagerank/vertex_push.cu:17:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 17 47
    mov.u32         %r29, %nctaid.x;
    mul.lo.s32      %r31, %r29, %r12;
    cvt.u64.u32     %rd51, %r31;
    add.s64         %rd52, %rd51, %rd52;

;pagerank/vertex_push.cu:17:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 17 5
    setp.lt.u64     %p9, %rd52, %rd2;
    @%p9 bra        BB0_2;

BB0_14:
;pagerank/vertex_push.cu:31:
;}
    .loc 1 31 1
    ret;
}
```

B.3 Vertex Pull

```
;
; Generated by NVIDIA NVVM Compiler
;
; Compiler Build ID: CL-24817639
; Cuda compilation tools, release 10.0, V10.0.130
; Based on LLVM 3.4svn

.version 6.3
.target sm_53
.address_size 64

.visible .entry vertexPullPageRank(
    .param .u64 vertexPullPageRank_graph,
    .param .u64 vertexPullPageRank_degrees,
    .param .u64 vertexPullPageRank_pagerank,
    .param .u64 vertexPullPageRank_new_pagerank
)
{
    .reg .pred      %p<9>;
    .reg .f32       %f<45>;
    .reg .b32       %r<41>;
    .reg .b64       %rd<65>;

    ld.param.u64    %rd8, [vertexPullPageRank_graph];
    ld.param.u64    %rd9, [vertexPullPageRank_degrees];
    ld.param.u64    %rd10, [vertexPullPageRank_pagerank];
    ld.param.u64    %rd11, [vertexPullPageRank_new_pagerank];

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd12, %rd8;
    mov.u32         %r13, %ntid.x;
    mov.u32         %r14, %ctaid.x;
    mov.u32         %r15, %tid.x;
    mad.lo.s32      %r16, %r13, %r14, %r15;
    cvt.u64.u32     %rd64, %r16;

;pagerank/vertex_pull.cu:12:
;uint64_t size = graph->vertex_count;
    .loc 1 12 19
    ld.global.u64   %rd2, [%rd12];
```

```
;pagerank/vertex_pull.cu:16:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 5
    setp.ge.u64     %p1, %rd64, %rd2;
    @%p1 bra        BB0_13;

    mov.f32         %f44, 0f00000000;

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd60, %rd11;

BB0_2:
;pagerank/vertex_pull.cu:17:
;unsigned *rev_vertices = graph->vertices;
    .loc 1 17 32
    ld.global.u64   %rd14, [%rd12+16];

;pagerank/vertex_pull.cu:18:
;unsigned *reverse_edges = graph->edges;
    .loc 1 18 33
    cvta.to.global.u64      %rd15, %rd14;
    ld.global.u64   %rd16, [%rd12+24];

;pagerank/vertex_pull.cu:20:
;unsigned start = rev_vertices[idx];
    .loc 1 20 24
    cvta.to.global.u64      %rd4, %rd16;
    shl.b64         %rd17, %rd64, 2;
    add.s64         %rd18, %rd15, %rd17;

;pagerank/vertex_pull.cu:21:
;unsigned end = rev_vertices[idx + 1];
    .loc 1 21 22
    ld.global.u32   %r1, [%rd18+4];

;pagerank/vertex_pull.cu:20:
;unsigned start = rev_vertices[idx];
    .loc 1 20 24
    ld.global.u32   %r2, [%rd18];
```

```
;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 9
    setp.le.u32     %p2, %r1, %r2;
    @%p2 bra        BB0_12;

    sub.s32         %r3, %r1, %r2;
    and.b32         %r4, %r3, 3;
    setp.eq.s32     %p3, %r4, 0;
    mov.f32         %f13, 0f00000000;
    @%p3 bra        BB0_4;
    bra.uni         BB0_5;

BB0_4:
    mov.f32         %f43, %f44;
    mov.f32         %f44, %f13;
    bra.uni         BB0_10;

BB0_5:
    setp.eq.s32     %p4, %r4, 1;
    @%p4 bra        BB0_9;

    setp.eq.s32     %p5, %r4, 2;
    @%p5 bra        BB0_8;

;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd19, %r2, 4;
    add.s64         %rd20, %rd4, %rd19;
    ld.global.u32   %r17, [%rd20];

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd21, %rd10;

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd22, %r17, 4;
    add.s64         %rd23, %rd21, %rd22;
```

```
;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd24, %rd9;

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    add.s64         %rd25, %rd24, %rd22;
    ld.global.u32   %r18, [%rd25];
    cvt.rn.f32.u32  %f14, %r18;
    ld.global.f32   %f15, [%rd23];
    div.rn.f32      %f16, %f15, %f14;
    add.f32         %f44, %f44, %f16;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;

BB0_8:
;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd26, %r2, 4;
    add.s64         %rd27, %rd4, %rd26;
    ld.global.u32   %r19, [%rd27];

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd28, %rd10;

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd29, %r19, 4;
    add.s64         %rd30, %rd28, %rd29;

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd31, %rd9;
```

```
;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    add.s64         %rd32, %rd31, %rd29;
    ld.global.u32   %r20, [%rd32];
    cvt.rn.f32.u32  %f17, %r20;
    ld.global.f32   %f18, [%rd30];
    div.rn.f32      %f19, %f18, %f17;
    add.f32         %f44, %f44, %f19;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;

BB0_9:
;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd33, %r2, 4;
    add.s64         %rd34, %rd4, %rd33;
    ld.global.u32   %r21, [%rd34];

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd35, %rd10;

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd36, %r21, 4;
    add.s64         %rd37, %rd35, %rd36;

;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd38, %rd9;

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    add.s64         %rd39, %rd38, %rd36;
    ld.global.u32   %r22, [%rd39];
```

```
    cvt.rn.f32.u32  %f20, %r22;
    ld.global.f32   %f21, [%rd37];
    div.rn.f32      %f22, %f21, %f20;
    add.f32         %f43, %f44, %f22;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;
    mov.f32         %f44, %f43;

BB0_10:
;pagerank/vertex_pull.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd5, %rd10;
    cvta.to.global.u64      %rd6, %rd9;
    setp.lt.u32     %p6, %r3, 4;
    @%p6 bra        BB0_12;

BB0_11:
;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd40, %r2, 4;
    add.s64         %rd41, %rd4, %rd40;
    ld.global.u32   %r23, [%rd41];

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd42, %r23, 4;
    add.s64         %rd43, %rd5, %rd42;
    add.s64         %rd44, %rd6, %rd42;
    ld.global.u32   %r24, [%rd44];
    cvt.rn.f32.u32  %f23, %r24;
    ld.global.f32   %f24, [%rd43];
    div.rn.f32      %f25, %f24, %f23;
    add.f32         %f26, %f43, %f25;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r25, %r2, 1;
```

```
;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd45, %r25, 4;
    add.s64         %rd46, %rd4, %rd45;
    ld.global.u32   %r26, [%rd46];

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd47, %r26, 4;
    add.s64         %rd48, %rd5, %rd47;
    add.s64         %rd49, %rd6, %rd47;
    ld.global.u32   %r27, [%rd49];
    cvt.rn.f32.u32  %f27, %r27;
    ld.global.f32   %f28, [%rd48];
    div.rn.f32      %f29, %f28, %f27;
    add.f32         %f30, %f26, %f29;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r28, %r2, 2;

;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd50, %r28, 4;
    add.s64         %rd51, %rd4, %rd50;
    ld.global.u32   %r29, [%rd51];

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd52, %r29, 4;
    add.s64         %rd53, %rd5, %rd52;
    add.s64         %rd54, %rd6, %rd52;
    ld.global.u32   %r30, [%rd54];
    cvt.rn.f32.u32  %f31, %r30;
    ld.global.f32   %f32, [%rd53];
    div.rn.f32      %f33, %f32, %f31;
    add.f32         %f34, %f30, %f33;
```

```
;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r31, %r2, 3;

;pagerank/vertex_pull.cu:24:
;uint64_t rev_edge = reverse_edges[i];
    .loc 1 24 31
    mul.wide.u32    %rd55, %r31, 4;
    add.s64         %rd56, %rd4, %rd55;
    ld.global.u32   %r32, [%rd56];

;pagerank/vertex_pull.cu:26:
;newRank += pagerank[rev_edge] / degrees[rev_edge];
    .loc 1 26 13
    mul.wide.u32    %rd57, %r32, 4;
    add.s64         %rd58, %rd5, %rd57;
    add.s64         %rd59, %rd6, %rd57;
    ld.global.u32   %r33, [%rd59];
    cvt.rn.f32.u32  %f35, %r33;
    ld.global.f32   %f36, [%rd58];
    div.rn.f32      %f37, %f36, %f35;
    add.f32         %f43, %f34, %f37;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 4;

;pagerank/vertex_pull.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 9
    setp.lt.u32     %p7, %r2, %r1;
    mov.f32         %f44, %f43;
    @%p7 bra        BB0_11;

BB0_12:
;pagerank/vertex_pull.cu:29:
;new_pagerank[idx] = newRank;
    .loc 1 29 9
    add.s64         %rd62, %rd60, %rd17;
    st.global.f32   [%rd62], %f44;

;pagerank/vertex_pull.cu:16:
```

```
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 47
    mov.u32         %r35, %nctaid.x;
    mul.lo.s32      %r36, %r35, %r13;
    cvt.u64.u32     %rd63, %r36;
    add.s64         %rd64, %rd63, %rd64;

;pagerank/vertex_pull.cu:16:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 5
    setp.lt.u64     %p8, %rd64, %rd2;
    @%p8 bra        BB0_2;

BB0_13:
;pagerank/vertex_pull.cu:31:
;}
    .loc 1 31 1
    ret;
}
```

B.4 Vertex Pull NoDiv

```
;
; Generated by NVIDIA NVVM Compiler
;
; Compiler Build ID: CL-24817639
; Cuda compilation tools, release 10.0, V10.0.130
; Based on LLVM 3.4svn

.version 6.3
.target sm_53
.address_size 64

.visible .entry vertexPullNoDivPageRank(
    .param .u64 vertexPullNoDivPageRank_graph,
    .param .u64 vertexPullNoDivPageRank_unused,
    .param .u64 vertexPullNoDivPageRank_pagerank,
    .param .u64 vertexPullNoDivPageRank_new_pagerank
)
```

```
{
    .reg .pred      %p<9>;
    .reg .f32       %f<31>;
    .reg .b32       %r<34>;
    .reg .b64       %rd<53>;

    ld.param.u64    %rd7, [vertexPullNoDivPageRank_graph];
    ld.param.u64    %rd8, [vertexPullNoDivPageRank_pagerank];
    ld.param.u64    %rd9, [vertexPullNoDivPageRank_new_pagerank];

;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd10, %rd7;
    mov.u32         %r13, %ntid.x;
    mov.u32         %r14, %ctaid.x;
    mov.u32         %r15, %tid.x;
    mad.lo.s32      %r16, %r13, %r14, %r15;
    cvt.u64.u32     %rd52, %r16;

;pagerank/vertex_pull_nodiv.cu:12:
;uint64_t size = graph->vertex_count;
    .loc 1 12 19
    ld.global.u64   %rd2, [%rd10];

;pagerank/vertex_pull_nodiv.cu:16:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 5
    setp.ge.u64     %p1, %rd52, %rd2;
    @%p1 bra        BB0_13;

    mov.f32         %f30, 0f00000000;

;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd48, %rd9;

BB0_2:
;pagerank/vertex_pull_nodiv.cu:17:
;unsigned *rev_vertices = &graph->vertices[idx];
    .loc 1 17 32
```

```
    ld.global.u64   %rd12, [%rd10+16];
    cvta.to.global.u64      %rd13, %rd12;
    shl.b64         %rd14, %rd52, 2;
    add.s64         %rd15, %rd13, %rd14;

;pagerank/vertex_pull_nodiv.cu:21:
;unsigned *rev_edges = graph->edges;
    .loc 1 21 29
    ld.global.u64   %rd16, [%rd10+24];

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 9
    cvta.to.global.u64      %rd4, %rd16;

;pagerank/vertex_pull_nodiv.cu:19:
;unsigned end = rev_vertices[1];
    .loc 1 19 22
    ld.global.u32   %r1, [%rd15+4];

;pagerank/vertex_pull_nodiv.cu:18:
;unsigned start = rev_vertices[0];
    .loc 1 18 24
    ld.global.u32   %r2, [%rd15];

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 9
    setp.le.u32     %p2, %r1, %r2;
    @%p2 bra        BB0_12;

    sub.s32         %r3, %r1, %r2;
    and.b32         %r4, %r3, 3;
    setp.eq.s32     %p3, %r4, 0;
    mov.f32         %f13, 0f00000000;
    @%p3 bra        BB0_4;
    bra.uni         BB0_5;

BB0_4:
    mov.f32         %f29, %f30;
    mov.f32         %f30, %f13;
    bra.uni         BB0_10;

BB0_5:
```

```
    setp.eq.s32     %p4, %r4, 1;
    @%p4 bra        BB0_9;

    setp.eq.s32     %p5, %r4, 2;
    @%p5 bra        BB0_8;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd17, %r2, 4;
    add.s64         %rd18, %rd4, %rd17;
    ld.global.u32   %r17, [%rd18];

;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd19, %rd8;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd20, %r17, 4;
    add.s64         %rd21, %rd19, %rd20;
    ld.global.f32   %f14, [%rd21];
    add.f32         %f30, %f30, %f14;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;

BB0_8:
;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd22, %r2, 4;
    add.s64         %rd23, %rd4, %rd22;
    ld.global.u32   %r18, [%rd23];

;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd24, %rd8;
```

```
;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd25, %r18, 4;
    add.s64         %rd26, %rd24, %rd25;
    ld.global.f32   %f15, [%rd26];
    add.f32         %f30, %f30, %f15;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;

BB0_9:
;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd27, %r2, 4;
    add.s64         %rd28, %rd4, %rd27;
    ld.global.u32   %r19, [%rd28];

;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
    cvta.to.global.u64      %rd29, %rd8;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd30, %r19, 4;
    add.s64         %rd31, %rd29, %rd30;
    ld.global.f32   %f16, [%rd31];
    add.f32         %f29, %f30, %f16;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 1;
    mov.f32         %f30, %f29;

BB0_10:
;pagerank/vertex_pull_nodiv.cu:11:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 11 23
```

```
    cvta.to.global.u64      %rd5, %rd8;
    setp.lt.u32     %p6, %r3, 4;
    @%p6 bra        BB0_12;

BB0_11:
;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd32, %r2, 4;
    add.s64         %rd33, %rd4, %rd32;
    ld.global.u32   %r20, [%rd33];
    mul.wide.u32    %rd34, %r20, 4;
    add.s64         %rd35, %rd5, %rd34;
    ld.global.f32   %f17, [%rd35];
    add.f32         %f18, %f29, %f17;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r21, %r2, 1;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd36, %r21, 4;
    add.s64         %rd37, %rd4, %rd36;
    ld.global.u32   %r22, [%rd37];
    mul.wide.u32    %rd38, %r22, 4;
    add.s64         %rd39, %rd5, %rd38;
    ld.global.f32   %f19, [%rd39];
    add.f32         %f20, %f18, %f19;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r23, %r2, 2;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd40, %r23, 4;
    add.s64         %rd41, %rd4, %rd40;
    ld.global.u32   %r24, [%rd41];
    mul.wide.u32    %rd42, %r24, 4;
```

```
    add.s64         %rd43, %rd5, %rd42;
    ld.global.f32   %f21, [%rd43];
    add.f32         %f22, %f20, %f21;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r25, %r2, 3;

;pagerank/vertex_pull_nodiv.cu:24:
;newRank += pagerank[rev_edges[i]];
    .loc 1 24 13
    mul.wide.u32    %rd44, %r25, 4;
    add.s64         %rd45, %rd4, %rd44;
    ld.global.u32   %r26, [%rd45];
    mul.wide.u32    %rd46, %r26, 4;
    add.s64         %rd47, %rd5, %rd46;
    ld.global.f32   %f23, [%rd47];
    add.f32         %f29, %f22, %f23;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 43
    add.s32         %r2, %r2, 4;

;pagerank/vertex_pull_nodiv.cu:23:
;for (unsigned i = start; i < end; i++) {
    .loc 1 23 9
    setp.lt.u32     %p7, %r2, %r1;
    mov.f32         %f30, %f29;
    @%p7 bra        BB0_11;

BB0_12:
;pagerank/vertex_pull_nodiv.cu:27:
;new_pagerank[idx] = newRank;
    .loc 1 27 9
    add.s64         %rd50, %rd48, %rd14;
    st.global.f32   [%rd50], %f30;

;pagerank/vertex_pull_nodiv.cu:16:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 47
```

```
    mov.u32         %r28, %nctaid.x;
    mul.lo.s32      %r29, %r28, %r13;
    cvt.u64.u32     %rd51, %r29;
    add.s64         %rd52, %rd51, %rd52;

;pagerank/vertex_pull_nodiv.cu:16:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 16 5
    setp.lt.u64     %p8, %rd52, %rd2;
    @%p8 bra        BB0_2;

BB0_13:
;pagerank/vertex_pull_nodiv.cu:29:
;}
    .loc 1 29 1
    ret;
}
```

B.5 Consolidate & Consolidate NoDiv

```
;
; Generated by NVIDIA NVVM Compiler
;
; Compiler Build ID: CL-24817639
; Cuda compilation tools, release 10.0, V10.0.130
; Based on LLVM 3.4svn

.version 6.3
.target sm_53
.address_size 64

.visible .entry consolidateRank(
    .param .u64 consolidateRank_size,
    .param .u64 consolidateRank_unused1,
    .param .u64 consolidateRank_pagerank,
    .param .u64 consolidateRank_new_pagerank,
    .param .u8  consolidateRank_unused2
)
{
    .reg .pred      %p<7>;
    .reg .f32       %f<13>;
```

```
    .reg .b32       %r<19>;
    .reg .f64       %fd<6>;
    .reg .b64       %rd<15>;


    ld.param.u64    %rd7, [consolidateRank_size];
    ld.param.u64    %rd8, [consolidateRank_pagerank];
    ld.param.u64    %rd9, [consolidateRank_new_pagerank];

;pagerank/pagerank.cu:129:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 129 23
    mov.u32         %r1, %ntid.x;
    mov.u32         %r7, %ctaid.x;
    mov.u32         %r2, %tid.x;
    mad.lo.s32      %r8, %r1, %r7, %r2;
    cvt.u64.u32     %rd14, %r8;

;pagerank/pagerank.cu:131:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 131 5
    setp.ge.u64     %p1, %rd14, %rd7;
    @%p1 bra        BB7_7;

;pagerank/pagerank.cu:132:
;float new_rank = ((1.0 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 132 24
    cvt.rn.f64.u64  %fd2, %rd7;
    mov.f64         %fd3, 0d3FC3333300000000;

;pagerank/pagerank.cu:132:
;float new_rank = ((1.0 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 132 24
    div.rn.f64      %fd1, %fd3, %fd2;

;pagerank/pagerank.cu:31:
;int lane = threadIdx.x % warpSize;
    .loc 1 31 14
    mov.u32         %r3, WARP_SZ;
    rem.u32         %r4, %r2, %r3;
```

```
;pagerank/pagerank.cu:131:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 131 47
    mov.u32         %r9, %nctaid.x;
    mul.lo.s32      %r10, %r9, %r1;
    cvt.u64.u32     %rd2, %r10;

;pagerank/pagerank.cu:129:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 129 23
    cvta.to.global.u64      %rd3, %rd9;
    cvta.to.global.u64      %rd4, %rd8;

BB7_2:
;pagerank/pagerank.cu:132:
;float new_rank = ((1.0 - dampening) / size)
;                + (dampening * new_pagerank[idx]);
    .loc 1 132 24
    shl.b64         %rd10, %rd14, 2;
    add.s64         %rd11, %rd3, %rd10;
    ld.global.f32   %f5, [%rd11];
    mul.f32         %f6, %f5, 0f3F59999A;
    cvt.f64.f32     %fd4, %f6;
    add.f64         %fd5, %fd1, %fd4;
    cvt.rn.f32.f64  %f7, %fd5;

;pagerank/pagerank.cu:133:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 133 23
    add.s64         %rd12, %rd4, %rd10;
    ld.global.f32   %f8, [%rd12];
    sub.f32         %f9, %f7, %f8;

;pagerank/pagerank.cu:133:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 133 25
    abs.f32         %f12, %f9;

;pagerank/pagerank.cu:135:
;pagerank[idx] = new_rank;
    .loc 1 135 9
```

```
    st.global.f32   [%rd12], %f7;
    mov.u32         %r11, 0;

;pagerank/pagerank.cu:136:
;new_pagerank[idx] = 0.0f;
    .loc 1 136 9
    st.global.u32   [%rd11], %r11;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;     ; offset > 0
;     ; offset /= 2) {
    .loc 1 33 5
    setp.lt.s32     %p2, %r3, 2;
    mov.u32         %r18, %r3;
    @%p2 bra        BB7_4;

BB7_3:
    mov.b32          %r12, %f12;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;     ; offset > 0
;     ; offset /= 2) {
    .loc 1 33 5
    shr.u32         %r13, %r18, 31;
    add.s32         %r14, %r18, %r13;
    shr.s32         %r6, %r14, 1;
    mov.u32         %r15, 31;
    mov.u32         %r16, -1;
    shfl.sync.down.b32      %r17|%p3, %r12, %r6, %r15, %r16;
    mov.b32          %f10, %r17;
    add.f32         %f12, %f12, %f10;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;     ; offset > 0
;     ; offset /= 2) {
    .loc 1 33 5
    setp.gt.s32     %p4, %r18, 3;
    mov.u32         %r18, %r6;
    @%p4 bra        BB7_3;

BB7_4:
```

```
;pagerank/pagerank.cu:37:
;if (lane == 0) atomicAdd(&diff, val);
    .loc 1 37 5
    setp.ne.s32     %p5, %r4, 0;
    @%p5 bra        BB7_6;

    .loc 4 77 10
    mov.u64         %rd13, diff;
    atom.global.add.f32     %f11, [%rd13], %f12;

BB7_6:
;pagerank/pagerank.cu:131:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 131 47
    add.s64         %rd14, %rd2, %rd14;

;pagerank/pagerank.cu:131:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 131 5
    setp.lt.u64     %p6, %rd14, %rd7;
    @%p6 bra        BB7_2;

BB7_7:
;pagerank/pagerank.cu:140:
}
    .loc 1 140 1
    ret;
}

.visible .entry consolidateRankNoDiv(
    .param .u64 consolidateRankNoDiv_size
    .param .u64 consolidateRankNoDiv_degrees,
    .param .u64 consolidateRankNoDiv_pagerank,
    .param .u64 consolidateRankNoDiv_new_pagerank,
    .param .u8 consolidateRankNoDiv_notLast
)
{
    .reg .pred      %p<14>;
    .reg .b16       %rs<3>;
    .reg .f32       %f<30>;
```

```
    .reg .b32       %r<34>;
    .reg .b64       %rd<28>;

    ld.param.u64    %rd11, [consolidateRankNoDiv_size];
    ld.param.u64    %rd12, [consolidateRankNoDiv_degrees];
    ld.param.u64    %rd13, [consolidateRankNoDiv_pagerank];
    ld.param.u64    %rd14, [consolidateRankNoDiv_new_pagerank];
    ld.param.s8     %rs1, [consolidateRankNoDiv_notLast];

;pagerank/pagerank.cu:151:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 151 23
    mov.u32         %r1, %ntid.x;
    mov.u32         %r11, %ctaid.x;
    mov.u32         %r2, %tid.x;
    mad.lo.s32      %r12, %r1, %r11, %r2;
    cvt.u64.u32     %rd26, %r12;

;pagerank/pagerank.cu:153:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 5
    setp.ge.u64     %p1, %rd26, %rd11;
    @%p1 bra        BB8_15;

;pagerank/pagerank.cu:154:
;float new_rank = ((1 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 154 24
    cvt.rn.f32.u64  %f13, %rd11;
    mov.f32         %f14, 0f3E199998;

;pagerank/pagerank.cu:154:
;float new_rank = ((1 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 154 24
    div.rn.f32      %f1, %f14, %f13;

;pagerank/pagerank.cu:31:
;int lane = threadIdx.x % warpSize;
    .loc 1 31 14
    mov.u32         %r3, WARP_SZ;
    rem.u32         %r4, %r2, %r3;
```

```
;pagerank/pagerank.cu:153:
;for ( uint64_t idx = startIdx
;    ; idx < size
;    ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 47
    mov.u32         %r13, %nctaid.x;
    mul.lo.s32      %r14, %r13, %r1;
    cvt.u64.u32     %rd2, %r14;

;pagerank/pagerank.cu:159:
;if (degree != 0 && notLast) new_rank = new_rank / degree;
    .loc 1 159 9
    and.b16         %rs2, %rs1, 255;
    setp.eq.s16     %p2, %rs2, 0;

;pagerank/pagerank.cu:151:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 151 23
    cvta.to.global.u64      %rd3, %rd14;
    cvta.to.global.u64      %rd4, %rd13;
    @%p2 bra        BB8_10;

;pagerank/pagerank.cu:151:
;uint64_t startIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    .loc 1 151 23
    cvta.to.global.u64      %rd18, %rd12;

BB8_3:
;pagerank/pagerank.cu:154:
;float new_rank = ((1 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 154 24
    shl.b64         %rd16, %rd26, 2;
    add.s64         %rd6, %rd3, %rd16;
    ld.global.f32   %f15, [%rd6];
    fma.rn.f32      %f27, %f15, 0f3F59999A, %f1;

;pagerank/pagerank.cu:155:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 155 23
    add.s64         %rd7, %rd4, %rd16;
    ld.global.f32   %f16, [%rd7];
    sub.f32         %f17, %f27, %f16;
```

```
;pagerank/pagerank.cu:155:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 155 25
    abs.f32         %f28, %f17;

;pagerank/pagerank.cu:157:
;unsigned degree = degrees[idx];
    .loc 1 157 25
    add.s64         %rd19, %rd18, %rd16;
    ld.global.u32   %r5, [%rd19];

;pagerank/pagerank.cu:159:
;if (degree != 0 && notLast) new_rank = new_rank / degree;
    .loc 1 159 9
    setp.eq.s32     %p3, %r5, 0;
    @%p3 bra        BB8_5;

;pagerank/pagerank.cu:159:
;if (degree != 0 && notLast) new_rank = new_rank / degree;
    .loc 1 159 37
    cvt.rn.f32.u32  %f18, %r5;
    div.rn.f32      %f27, %f27, %f18;

BB8_5:
;pagerank/pagerank.cu:31:
;int lane = threadIdx.x % warpSize;
    .loc 1 31 14
    mov.u32         %r32, WARP_SZ;

;pagerank/pagerank.cu:160:
;pagerank[idx] = new_rank;
    .loc 1 160 9
    st.global.f32   [%rd7], %f27;
    mov.u32         %r15, 0;

;pagerank/pagerank.cu:161:
;new_pagerank[idx] = 0.0f;
    .loc 1 161 9
    st.global.u32   [%rd6], %r15;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
```

```
;    ; offset /= 2) {
    .loc 1 33 5
    setp.lt.s32     %p4, %r32, 2;
    @%p4 bra        BB8_7;

BB8_6:
    mov.b32          %r16, %f28;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
;    ; offset /= 2) {
    .loc 1 33 5
    shr.u32         %r17, %r32, 31;
    add.s32         %r18, %r32, %r17;
    shr.s32         %r8, %r18, 1;
    mov.u32         %r19, 31;
    mov.u32         %r20, -1;
    shfl.sync.down.b32      %r21|%p5, %r16, %r8, %r19, %r20;
    mov.b32          %f19, %r21;
    add.f32         %f28, %f28, %f19;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
;    ; offset /= 2) {
    .loc 1 33 5
    setp.gt.s32     %p6, %r32, 3;
    mov.u32         %r32, %r8;
    @%p6 bra        BB8_6;

BB8_7:
;pagerank/pagerank.cu:37:
;if (lane == 0) atomicAdd(&diff, val);
    .loc 1 37 5
    setp.ne.s32     %p7, %r4, 0;
    @%p7 bra        BB8_9;

    .loc 4 77 10
    mov.u64         %rd20, diff;
    atom.global.add.f32     %f20, [%rd20], %f28;

BB8_9:
;pagerank/pagerank.cu:153:
```

```
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 47
    add.s64         %rd26, %rd2, %rd26;

;pagerank/pagerank.cu:153:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 5
    setp.lt.u64     %p8, %rd26, %rd11;
    @%p8 bra        BB8_3;
    bra.uni         BB8_15;

BB8_10:
;pagerank/pagerank.cu:154:
;float new_rank = ((1 - dampening) / size)
;               + (dampening * new_pagerank[idx]);
    .loc 1 154 24
    shl.b64         %rd22, %rd26, 2;
    add.s64         %rd23, %rd3, %rd22;
    ld.global.f32   %f21, [%rd23];
    fma.rn.f32      %f22, %f21, 0f3F59999A, %f1;

;pagerank/pagerank.cu:155:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 155 23
    add.s64         %rd24, %rd4, %rd22;
    ld.global.f32   %f23, [%rd24];
    sub.f32         %f24, %f22, %f23;

;pagerank/pagerank.cu:155:
;float my_diff = fabsf(new_rank - pagerank[idx]);
    .loc 1 155 25
    abs.f32         %f29, %f24;

;pagerank/pagerank.cu:160:
;pagerank[idx] = new_rank;
    .loc 1 160 9
    st.global.f32   [%rd24], %f22;
    mov.u32         %r25, 0;

;pagerank/pagerank.cu:161:
```

```
;new_pagerank[idx] = 0.0f;
    .loc 1 161 9
    st.global.u32   [%rd23], %r25;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
;    ; offset /= 2) {
    .loc 1 33 5
    setp.lt.s32     %p9, %r3, 2;
    mov.u32         %r33, %r3;
    @%p9 bra        BB8_12;

BB8_11:
    mov.b32          %r26, %f29;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
;    ; offset /= 2) {
    .loc 1 33 5
    shr.u32         %r27, %r33, 31;
    add.s32         %r28, %r33, %r27;
    shr.s32         %r10, %r28, 1;
    mov.u32         %r29, 31;
    mov.u32         %r30, -1;
    shfl.sync.down.b32      %r31|%p10, %r26, %r10, %r29, %r30;
    mov.b32          %f25, %r31;
    add.f32         %f29, %f29, %f25;

;pagerank/pagerank.cu:33:
;for ( int offset = warpSize/2
;    ; offset > 0
;    ; offset /= 2) {
    .loc 1 33 5
    setp.gt.s32     %p11, %r33, 3;
    mov.u32         %r33, %r10;
    @%p11 bra       BB8_11;

BB8_12:
;pagerank/pagerank.cu:37:
;if (lane == 0) atomicAdd(&diff, val);
    .loc 1 37 5
    setp.ne.s32     %p12, %r4, 0;
```

```
    @%p12 bra       BB8_14;

    .loc 4 77 10
    mov.u64         %rd25, diff;
    atom.global.add.f32     %f26, [%rd25], %f29;

BB8_14:
;pagerank/pagerank.cu:153:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 47
    add.s64         %rd26, %rd2, %rd26;

;pagerank/pagerank.cu:153:
;for ( uint64_t idx = startIdx
;     ; idx < size
;     ; idx += blockDim.x * gridDim.x) {
    .loc 1 153 5
    setp.lt.u64     %p13, %rd26, %rd11;
    @%p13 bra       BB8_10;

BB8_15:
;pagerank/pagerank.cu:165:
;}
    .loc 1 165 1
    ret;
}
```

Summary

Analysis and Prediction of GPU Graph Algorithm Performance

Accelerator devices in general, and Graphical Processing Units (GPUs) in particular, have become a staple of High-Performance Computing (HPC) systems. This means we need to be able to analyse, understand, and predict the performance of General Processing on GPU (GPGPU) code to effectively utilise these systems.

For regular algorithms, i.e. algorithms with static memory access patterns, a lot of research was successful in demonstrating how they can be mapped to GPGPUs, how their performance can be analysed, and how we can predict their performance.

This is not the case for irregular algorithms — i.e., algorithms whose access patterns are not known at compile time and/or depend on the input data. Combining modern GPU architectures — which rely on a rigid, regular architecture and deep memory hierarchies for their massive parallelism — with irregular algorithms, which are dynamic in nature, is challenging. Consequently, performance optimization and prediction for GPU irregular algorithms remains difficult.

In this thesis, we focus on a subset of irregular algorithms: graph processing algorithms. These are common in many scientific fields, due to the flexibility of graphs as a model for highly interrelated data. However, they are also the poster child of irregular algorithms, because their execution is entirely dependent on the structure of the input graph.

A number of state-of-the-art GPU graph processing frameworks [17, 37, 45, 49, 66, 69, 97, 102] have been created to tackle the challenge of high-performance graph processing. Each of these frameworks features its own primitives, techniques, and optimisations to handle the irregularity present in input graphs. The authors go to great length to explain how their primitives can be implemented efficiently on the GPU and show how fast their framework performs.

However, there is not a lot of information on how we can understand *why* these techniques get the performance that they do. Nor are there clear results on *how* this performance relates to the input data or the hardware being used.

To begin to understand the performance of GPU graph processing, we need a holistic approach that analyses the complex interaction between hardware, input data, algorithm, and data structures. This, in turn, requires a systematic overview of the data along these different dimensions. The state-of-the-art lacks the tools, techniques, theories, and workflows needed to get such an overview.

The problem space of graph processing on GPUs is too large, too intertwined, and too diverse to tackle in one thesis. As such, the main goal of the research discussed in this thesis is threefold:

1. To quantify the performance impact of structural properties of graphs in relation to data structure choices and hardware platforms.

2. To provide tools and a workflow that provide a systematic overview of how input data, algorithm, data structure, and hardware affect each other.

3. To investigate whether it is feasible to use these results to improve the current performance of graph processing on GPUs.

In Chapter 3 on page 17 we present the high-level architecture of the software pipeline we built to setup, collect, aggregate, and analyse the performance data for comprehensive GPU graph processing evaluation. Using a single toolchain and a database to pack and manage all the data and metadata improves usability, and simplifies the tasks needed to reproduce our work, and helps track the provenance of our results.

Furthermore, the single file format and wide support for SQLite make it trivial to share entire result sets with other researchers, letting them build their own work on top of the existing results without having to redo all the time consuming benchmarks themselves.

In Chapter 4 on page 41 we investigate how the performance of different parallelisation strategies for neighbour iteration changes across input graphs. Neighbour iteration appears as a primitive graph operation in many algorithms, making these results applicable beyond the PageRank and Breadth-First Search (BFS) algorithms we use for this investigation.

We show that there is a significant variation — for both BFS and PageRank — in the performance of each parallelisation strategy across different input graphs, up to several orders of magnitude. Furthermore, we also demonstrate that the performance of BFS does not just vary with the input graph, but also with the stage of the BFS traversal. We show

that correctly predicting the best implementation for each BFS step can produce significant performance gains.

To better illustrate the correlation between graph properties and parallel performance, we investigated controlled graph generation — i.e., generating graphs with specific properties. In Chapter 5 on page 67 we present our graph generator, focusing on the design of the *evolutionary computing approach* it uses. While our graph generator was successful at creating small-scale graphs with controlled properties, it failed to scale up to the larger graph sizes we need to draw any conclusions about the link between graph structure and parallelisation.

In Chapter 6 on page 85 we present workload models for each of our PageRank parallelisation strategies. These analytical workload models are based on memory accesses, as PageRank is largely memory-bound. We validated our models against the behaviour observed by NVIDIA's profiling tools, and observed strong agreement between modelled and measured data.

We show that, despite this match between models and profiling data, our workload models are not sufficient to predict the fastest implementation for a given graph. Further experiments — with different graph orderings — also show that is not even possible to statically approximate the parallel execution of our workload models. We conclude that accurately predicting the performance of the different parallelisation strategies requires modelling the runtime behaviour of the GPU, and is thus not feasible.

In Chapter 7 on page 99 we used the 247 graphs from the KONECT [51] dataset to collect performance data on different systems. We used this data to train and test a Binary Decision Tree (BDT) model to predict the best parallelisation strategy for a given graph. We show that our BDT model's predictions result outperform all of the static implementations in our dataset.

In Chapter 8 on page 111 we show that our BDT models are not simply memorising the training data. We prove this by varying the size of the training set used to train our models and show that models trained on a fraction of our result set are still effective.

We also show that our BDT models are not limited to the dataset they were trained on. In turn, this brings empirical evidence that our models capture — part of — the link between graph structure and parallelisation strategy, rather than memorising results from a specific result set.

In Chapter 9 on page 125 we conclude that this thesis is really a number of starting points. We showed the performance impact of structural properties of graphs is significant (see Chapter 4), that this impact is exploitable (see Chapter 7), and, finally, that this impact is consistent across datasets and GPU architectures (see Chapter 8). However, none of these things are directly applicable in "the real world" on their own. The main

takeaway of this thesis is the software toolchain and, to a lesser extent, the dataset we built to investigate and produce these results.

Both the toolchain and dataset have enormous potential for reuse and further research. One natural extension of the work in this thesis is to generalise it to additional algorithms, such as Single-Source Shortest Path (SSSP) and Betweenness Centrality (BC). Another direction would be to explore even more datasets of graphs, comparing the behaviour of our algorithm implementations across datasets.

It would be interesting to perform a more comprehensive investigation into the effect of different in memory orderings of graphs, expanding on our initial exploration in Chapter 6. This investigation can focus on how the quality of our BDT models is affected by these reorderings. Another potential research application of our BDT models could be to repurpose them as rudimentary (structural) graph classification schemes.

Samenvatting

Ontleding en Voorspelling van de Prestaties van Graafalgoritmes op Beeldverwerkingseenheden

Versnellingsapparatuur in het algemeen, en dan bij uitstek Beeldverwerkingseenheden (BVE's), zijn een standaard onderdeel van hoogprestatieberekeningssystemen geworden. Dit betekent dat het ontleden, begrijpen, en voorspellen van de prestaties van algemeengebruiksberekeningen op BVE's van het hoogste belang is om dergelijke hoogprestatieberekeningssystemen naar behoren te kunnen gebruiken.

Voor regelmatige algoritmes — dat wil zeggen, algoritmes met onveranderlijke, van tevoren bekende geheugenhandelingspatronen — is er een weelde aan onderzoek dat toont hoe deze algoritmes op een BVE uitgevoerd kunnen worden en hoe hun prestaties geanalyseerd en voorspeld kunnen worden.

Dit is niet het geval voor onregelmatige algoritmes — dat wil zeggen, algoritmes wiens geheugenhandelingspatronen niet van tevoren bekend zijn omdat zij, bijvoorbeeld, afhangen van de invoergegevens van een berekening. Moderne BVE's maken gebruik van een diepe geheugenhiërarchie en regelmatige verwerkingseenheidsstructuur om grootschalig parallellisme te bewerkstelligen. De veranderlijkheid van onregelmatige algoritmes maakt het uitdagend om deze doeltreffend uit te voeren op BVE's. Het verbeteren en voorspellen van de prestaties van onregelmatige algoritmes blijft dan ook ingewikkeld.

In dit proefschrift leggen we ons toe op een deelverzameling van de onregelmatige algoritmes: graafverwerkingsalgoritmes. Door de toepasbaarheid van grafen als model voor sterk onderlingverbonden gegevens, zijn zij veelvoorkomend in verscheidene wetenschappen. Tegelijkertijd, vormen grafen ook *het* klassieke voorbeeld voor onregelmatige algoritmes, doordat

de verwerking van grafen volkomen afhankelijk is van de opbouw van de invoergraaf.

Er zijn verschillende moderne voorbeelden van BVE-graafverwerkingsprogrammatuur [17, 37, 45, 49, 66, 69, 97, 102] die ontwikkeld zijn om de uitdagingen van hogesnelheidsgraafverwerking de baas te worden. Elk van deze voorbeelden gebruikt andere grondbeginselen, technieken, en verbeteringen om om te gaan met de onregelmatige structuur van invoergrafen. De schrijvers doen hun uiterste best om aan te tonen dat hun grondbeginselen doeltreffend op een BVE verwezenlijkt kunnen worden en dat dit leidt goede prestaties.

Er is echter weinig informatie die ons helpt begrijpen waar de prestaties geleverd door deze technieken en grondbeginselen vandaan komen. Noch is het duidelijk hoe deze prestaties in verband staan met de gebruikte invoergegevens en de onderliggende apparaten.

Om de prestaties van graafverwerking op BVE's te beginnen te begrijpen hebben we een holistische aanpak nodig die ons de ingewikkelde wisselwerking tussen invoergegevens, datastructuren, en onderliggende apparaten laat ontleden. Dit vereist een stelselmatig overzicht van alle informatie langs deze assen. De hulpmiddelen, technieken, theorieën, en werkwijzen die nodig zijn om zo'n overzicht te verkrijgen, ontbreken op dit moment.

Het onderzoeksgebied van graafverwerking op BVE's is te breed, te zeer met zichzelf verweven, en te verscheiden om in een proefschrift af te handelen. Hierom is het hoofddoel van dit proefschrift drievoudig:

1. Onderzoeken hoe de opbouw van invoergrafen de prestaties van graafverwerkingsalgoritmes beïnvloedt ten opzichte van datastructuurkeuzes en de onderliggende apparaten.

2. Ontwikkelen van hulpmiddelen en werkwijzen die een stelselmatig overzicht kunnen geven van de wisselwerking tussen invoergegevens, algoritmes, datastructuren, en onderliggende apparaten.

3. Onderzoeken of het haalbaar is om deze resultaten te gebruiken om de prestaties van graafverwerking op BVE's te verbeteren.

In Hoofdstuk 3 op pagina 17 tonen we een overzicht van het ontwerp van de door ons ontwikkelde programmatuur voor het opzetten, verzamelen, samenvoegen, en ontleden van prestatiegegevens voor een alomvattende evaluatie van BVE-graafverwerking. Het gebruik van een enkele gegevensbank voor beheer en opslag van alle gegevens en metadata verbetert de gebruikersvriendelijkheid, helpt de herkomst van gegevens te volgen, en versimpelt de reproductie van onze resultaten. Tevens betekent het gebruik van een enkel bestand, in het breedondersteunde SQLite formaat, maakt het delen van resultaten met andere onderzoekers makkelijker. Dit stelt andere onderzoekers in staat om eenvoudig door te bouwen op onze

resultaten, zonder deze tijdrovende prestatiebepalingen te hoeven herhalen.

In Hoofdstuk 4 op pagina 41 onderzoeken we hoe de prestaties van verschillende parallellisatiewijzen voor burenbezoek beïnvloed worden door de opbouw van invoergrafen. Burenbezoek is een basishandeling die in vele graafverwerkingsalgoritmes gebruikt wordt. Onze resultaten zijn dus ook bruikbaar buiten de PageRank en Breedte-eerst Zoekopdracht (BEZ) algoritmes die we voor dit onderzoek gebruiken.

We laten zie dat er betekenisvolle verschillen zijn — voor zowel BEZ en PageRank — in de prestaties van elke parallellisatiewijze. Afhankelijk van de invoergraaf kunnen deze verschillen oplopen tot meerdere ordes van grootte. Tevens tonen we aan dat de prestaties van BEZ niet alleen verschillen tussen invoergrafen, maar ook tussen stappen van het BEZ-verloop. We laten zien dat het juist voorspellen van de beste parallellisatiewijze voor een BEZ-stap betekenisvolle verbeteringen en prestaties oplevert.

Om de samenhang tussen graafeigenschappen en parallellisatieprestaties te verduidelijken, hebben we de gestuurde vervaardiging van grafen onderzocht — dat wil zeggen, de vervaardiging van graven met specifieke eigenschappen. In Hoofdstuk 5 op pagina 67 tonen we onze graafvervaardiger, met focus op het ontwerp van onze aanpak met een *evolutionair algoritme*. Hoewel onze graafvervaardiger succesvol kleinschalige grafen kan maken, is deze niet in staat dit te bewerkstelligen op de grotere schaal die we nodig hebben om conclusies te trekken over het verband tussen graafopbouw en parallellisatiewijzen.

In Hoofdstuk 6 op pagina 85 geven we werklastmodellen voor elk van onze PageRank parallellisatiewijzen. Deze analytische werklast modellen zijn gebaseerd op geheugenhandelingen, omdat PageRank grotendeels geheugengebonden is. We toetsen onze werklast modellen door ze te vergelijken met het gedrag waargenomen door NVIDIA's prestatiemetingsprogrammatuur. We zien een sterke overeenkomst tussen onze modellen en het waargenomen gedrag.

We laten zien dat, ondanks de overeenkomst tussen modellen en waarnemingen, onze werklast modellen niet voldoende zijn om voor een gegeven invoergraaf te voorspellen wat de parallellisatiewijze is die de beste prestaties oplevert. Verdere experimenten — met herordende versies van de grafen — tonen aan dat het niet mogelijk is om de parallelle uitvoer van onze werklast modellen te benaderen. We leiden hieruit af dat het nauwkeurig voorspellen van de prestaties van verschillende parallellisatiewijzen niet mogelijk is zonder een gedetailleerd model van het uitvoergedrag van de BVE en daarom niet redelijkerwijs haalbaar is.

In Hoofdstuk 7 op pagina 99 gebruiken we 247 grafen van de KONECT-dataset [51] en verzamelen prestatiemetingen van meerdere BVE-systemen. We gebruiken deze metingen om een Binaire Beslissingsboom (BBB) te trainen om de beste parallellisatiewijze voor een gegeven graaf te voorspellen.

We laten zien dat de voorspellingen van ons BBB-model beter presteren dan de onze statische parallellisatiewijzen.

In Hoofdstuk 8 op pagina 111 laten we zien dat onze BBB-modellen niet simpelweg de resultaten van onze metingen onthouden. We tonen dit aan de door trainingsinvoer van verschillende groottes te gebruiken voor het trainen van onze modellen. We laten zien dat modellen die slechts op een klein onderdeel van onze metingen getraind zijn, nog steeds doeltreffend zijn.

We laten ook zien dat onze BBB-modellen niet beperkt zijn tot de dataset waarop zij getraind zijn. Dit ondersteunt het idee dat onze modellen een deel van het verband tussen graafopbouw en parallellisatiegedrag vastleggen, en niet slechts de resultaten van een specifieke dataset onthouden.

In Hoofdstuk 9 op pagina 125 komen we tot de slotsom dat dit proefschrift een aantal verschillende beginpunten voor verder onderzoek biedt. We hebben laten zien dat de opbouw van een graaf een merkbare invloed heeft op prestaties (zie Hoofdstuk 4), dat deze invloed bruikbaar is (zie Hoofdstuk 7), en, tenslotte, dat deze invloed consistent is tussen datasets en BVE-ontwerpen (zie Hoofdstuk 8). Tegelijkertijd zijn geen van deze resultaten direct bruikbaar in “de echte wereld”. De belangrijkste bijdragen van dit proefschrift zijn de programmatuur die door ons ontwikkeld is en, in mindere mate, de dataset van resultaten die wij verzameld hebben.

De programmatuur en dataset bieden beide eindeloze mogelijkheden voor hergebruik en verder onderzoek. Een natuurlijke uitbreiding van het werk in dit proefschrift is om meer algoritmes, zoals het kortstepad-algoritme of het tussencentraliteit-algoritme op eenzelfde manier te ontleden. Een andere richting zou zijn om nog meer datasets van grafen te verkennen en gedragsverschillen van onze algoritmeïmplementaties te vergelijken tussen de verschillende datasets.

Het zou ook interessant zijn om een meer alomvattend onderzoek te doen naar het gevolg van verschillende mogelijke geheugenordeningen van grafen; voortbouwend op het verkennend onderzoek in Hoofdstuk 6. Dit onderzoek kan zich richten op hoe de verschillende herordeningen de kwaliteit van de voorspellingen van onze BBB-modellen aantast. Een andere mogelijke toepassing van onze BBB-modellen is om deze te hergebruiken als eenvoudige graafindelingsmethodes.

Publications

Merijn Verstraaten. *Belewitte*. Version 1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6959684. URL: https://doi.org/10.5281/zenodo.6959684.

Merijn Verstraaten. *Belewitte GPU Experiment Results*. Version v1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6925023. URL: https://doi.org/10.5281/zenodo.6925023.

Merijn Verstraaten. *Mix-and-Match Dataset*. Zenodo, Oct. 2018. DOI: 10.5281/zenodo.4317449. URL: https://doi.org/10.5281/zenodo.4317449.

Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Mix-and-Match: A Model-driven Runtime Optimisation Strategy for BFS on GPUs". In: *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE. 2018, pp. 53–60.

Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Quantifying the Performance Impact of Graph Structure on Neighbour Iteration Strategies for PageRank". In: *"Euro-Par 2015: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2015, pp. 528–540. ISBN: "978-3-319-27308-2".

Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Synthetic Graph Generation for Systematic Exploration of Graph Structural Properties". In: *"Euro-Par 2016: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2016, pp. 557–570. ISBN: "978-3-319-58943-5".

Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. *Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach*. 2017. eprint: arXiv:1708.01159.

Bibliography

[1] Maksudul Alam, Maleq Khan, Anil Vullikanti, and Madhav Marathe. "An efficient and scalable algorithmic method for generating large: scale random graphs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2016, p. 32.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. "Internet: Diameter of the world-wide web". In: *nature* 401.6749 (1999), p. 130.

[3] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, Henry A Gabb, Gurbinder Gill, Balint Hegyi, Scott Kolodziej, Tze Meng Low, Andrew Lumsdaine, Tugsbayasgalan Manlaibaatar, Timothy G Mattson, Scott McMillan, Ramesh Peri, Keshav Pingali, Upasana Sridhar, Gabor Szarnyas, Yunming Zhang, and Yongzhe Zhang. "Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite". In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020, pp. 216–227. DOI: 10.1109/IISWC50251.2020.00029.

[4] Benjamin Bach, Andre Spritzer, Evelyne Lutton, and Jean-Daniel Fekete. "Interactive random graph generation with evolutionary algorithms". In: *Graph Drawing*. Springer. 2013, pp. 541–552.

[5] Alexander Bailey, Mario Ventresca, and Beatrice Ombuki-Berman. "Automatic generation of graph models for complex networks by genetic programming". In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM. 2012, pp. 711–718.

[6] Monya Baker. "1,500 scientists lift the lid on reproducibility". In: *Nature News* 533.7604 (2016), p. 452.

[7] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. "A medium-scale distributed system for computer science research: Infrastructure for the long term". In: *Computer* 5 (2016), pp. 54–63.

[8] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: http://science.sciencemag.org/content/286/5439/509.full.pdf. URL: http://science.sciencemag.org/content/286/5439/509.

[9] Scott Beamer, Krste Asanović, and David Patterson. "Direction-optimizing breadth-first search". In: *Scientific Programming* 21.3-4 (2013), pp. 137–148.

[10] Ronald F Boisvert, Ronald F Boisvert, and Karin A Remington. *The Matrix Market Exchange Formats: Initial Design*. Vol. 5935. US Department of Commerce, National Institute of Standards and Technology, 1996.

[11] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and Regression Trees*. CRC press, 1984.

[12] Gunnar Brinkmann, Kris Coolsaet, Jan Goedgebeur, and Hadrien Mélot. "House of Graphs: a database of interesting graphs". In: *Discrete Applied Mathematics* 161.1-2 (2013), pp. 311–314.

[13] Aydın Buluç. "Linear Algebraic Primitives for Parallel Computing on Large Graphs". PhD thesis. University of California, Santa Barbara, 2010.

[14] Aydın Buluç, Scott Beamer, Kamesh Madduri, Krste Asanović, and David Patterson. "Distributed-Memory Breadth-First Search on Massive Graphs". In: *Parallel Graph Algorithms*. Ed. by D. Bader. CRC Press, Taylor-Francis, 2016. URL: http://gauss.cs.ucsb.edu/~aydin/ChapterBFS2015.pdf.

[15] Aydın Buluç, John R. Gilbert, and Ceren Budak. "Solving path problems on the GPU". In: *Parallel Computing* 36.5-6 (2010), pp. 241–253. DOI: 10.1016/j.parco.2009.12.002. URL: http://gauss.cs.ucsb.edu/publication/parco_apsp.pdf.

[16] Aydın Buluç, John R. Gilbert, and Viral B. Shah. "Implementing Sparse Matrices for Graph Algorithms". In: *Graph Algorithms in the Language of Linear Algebra*. Ed. by Jeremy Kepner and John R. Gilbert. SIAM Press, 2011. DOI: 10.1137/1.9780898719918.ch13.

[17] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs". In: *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 141–151.

[18] CERN. *Zenodo*. URL: https://zenodo.org/ (visited on 16-10-2020).

[19] Deepayan Chakrabarti and Christos Faloutsos. "Graph mining: Laws, generators, and algorithms". In: *ACM Computing Surveys (CSUR)* 38.1 (2006), p. 2.

[20] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining." In: *SDM*. Vol. 4. SIAM. 2004, pp. 442–446.

[21] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[22] Chris J Cheney. "A nonrecursive list compacting algorithm". In: *Communications of the ACM* 13.11 (1970), pp. 677–678.

[23] Christian Collberg, Todd Proebsting, and Alex M Warren. "Repeatability and benefaction in computer systems research". In: *University of Arizona TR* 14.4 (2015).

[24] The Graph 500 Steering Committee. *The Graph 500 List*. 2010. URL: http://www.graph500.org.

[25] Library of Congres. *Recommended Formats Statement*. URL: https://www.loc.gov/preservation/resources/rfs/data.html (visited on 10-12-2018). Archive link: https://web.archive.org/web/20181206134744/https://www.loc.gov/preservation/resources/rfs/data.html.

[26] Creative Commons. *Creative Commons Attribution 4.0 International Public License*. Nov. 1, 2013. URL: https://creativecommons.org/licenses/by/4.0/legalcode (visited on 13-12-2020). Archive link: https://web.archive.org/web/20201213114731/https://creativecommons.org/licenses/by/4.0/legalcode.

[27] Peter J Denning. "ACM President's Letter: What is Experimental Computer Science?" In: *Communications of the ACM* 23.10 (1980), pp. 543–544.

[28] Peter J Denning. "The Profession of IT, Is Computer Science Science?" In: *Communications of the ACM* 48.4 (2005), pp. 27–31.

[29] Stephan Druskat, Jurriaan H. Spaaks, Neil Chue Hong, Robert Haines, and James Baker. *Citation File Format (CFF) - Specifications*. Version 1.0.3-4. Nov. 2019. DOI: 10.5281/zenodo.3515946. URL: https://doi.org/10.5281/zenodo.3515946.

[30] Paul Erdös and Alfréd Rényi. "{On the evolution of random graphs}". In: *Publ. Math. Inst. Hung. Acad. Sci* 5 (1960), pp. 17–61.

[31] Leonhard Euler. “Solutio problematis ad geometriam situs pertinensis”. In: *Commentarii academiae scientiarum Petropolitanae* 8 (1736), pp. 128–140.

[32] Lester Randolph Ford and Delbert R Fulkerson. “Maximal flow through a network”. In: *Canadian journal of Mathematics* 8 (1956), pp. 399–404.

[33] The Apache Software Foundation. *Apache Giraph*. May 25, 2018. URL: https://giraph.apache.org/ (visited on 18-12-2018). Archive link: https://web.archive.org/web/20181207082238/https://giraph.apache.org/.

[34] Free Software Foundation. *GNU General Public License*. June 29, 2007. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 6-12-2018). Archive link: https://web.archive.org/web/20181205113612/https://www.gnu.org/licenses/gpl-3.0.en.html.

[35] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. “Communication-free massively distributed graph generation”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 336–347.

[36] Jason Gauci and Kenneth O Stanley. “Autonomous evolution of topographic regularities in artificial neural networks”. In: *Neural computation* 22.7 (2010), pp. 1860–1898.

[37] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. “On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest”. In: *IPDPS*. 2013, pp. 851–862.

[38] Wolfgang K Giloi. “Konrad Zuse’s Plankalkuel: The First High-Level, ”non von Neumann” Programming Language”. In: *IEEE Annals of the History of Computing* 19.2 (1997), pp. 17–24.

[39] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. “Breaking the GPU programming barrier with the auto-parallelising SAC compiler”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 15–24.

[40] Mark Harris. *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. Apr. 22, 2013. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (visited on 26-10-2020). Archive link: https://web.archive.org/web/20201026132227/https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[41] Mark Harris. “High Performance Computing With CUDA”. In: 2007.

[42] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 556–571.

[43] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. “Green-Marl: a DSL for easy and efficient graph analysis”. In: *ACM SIGARCH Computer Architecture News*. Vol. 40. 1. ACM. 2012, pp. 349–362.

[44] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. “PGX.D: A Fast Distributed Graph Processing Engine”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 58.

[45] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. “Accelerating CUDA Graph Algorithms at Maximum Warp”. In: *ACM SIGPLAN Notices*. Vol. 46. 8. ACM. 2011, pp. 267–276.

[46] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. “LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms”. In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1317–1328. ISSN: 2150-8097. DOI: 10.14778/3007263.3007270. URL: https://doi.org/10.14778/3007263.3007270.

[47] Morris A. Jette, Andy B. Yoo, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[48] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. “Construction and use of linear regression models for processor performance analysis”. In: *International Symposium on High-Performance Computer Architecture*. 2006, pp. 99–108. DOI: 10.1109/HPCA.2006.1598116.

[49] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. “CuSha: vertex-centric graph processing on GPUs”. In: *HPCS*. ACM. 2014, pp. 239–252.

[50] Jérôme Kunegis. “Handbook of Network Analysis [KONECT - the Koblenz Network Collection]”. In: *CoRR* abs/1402.5500 (2014). arXiv: 1402.5500. URL: http://arxiv.org/abs/1402.5500.

[51] Jérôme Kunegis. “KONECT: The Koblenz Network Collection”. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13 Companion. Rio de Janeiro, Brazil, 2013, pp. 1343–1350. ISBN: 978-1-4503-2038-2.

[52] Anna-Lena Lamprecht, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, Stephanie van de Sandt, Jon Ison, Paula Andrea Martinez, et al. “Towards FAIR principles for research software”. In: *Data Science* Preprint (2019), pp. 1–23.

[53] E. Scott Larsen and David McAllister. “Fast Matrix Multiplies Using Graphics Hardware”. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC ’01. Denver, Colorado: ACM, 2001, pp. 55–55. ISBN: 1-58113-293-X. DOI: 10.1145/582034.582089. URL: http://doi.acm.org/10.1145/582034.582089.

[54] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. “Methods of Inference and Learning for Performance Modeling of Parallel Applications”. In: *PPoPP’07*. San Jose, California, USA: ACM, 2007. ISBN: 978-1-59593-602-8. DOI: 10.1145/1229428.1229479. URL: http://doi.acm.org/10.1145/1229428.1229479.

[55] Chin Yang Lee. “An algorithm for path connections and its applications”. In: *IRE transactions on electronic computers* 3 (1961), pp. 346–365.

[56] J. Leskovec. “Stanford Network Analysis Platform (SNAP)”. In: *Stanford University* (2006).

[57] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. “Kronecker graphs: An approach to modeling networks”. In: *The Journal of Machine Learning Research* 11 (2010), pp. 985–1042.

[58] D. Li and M. Becchi. “Deploying Graph Algorithms on GPUs: An Adaptive Solution”. In: *IPDPS 2013*. May 2013, pp. 1013–1024. DOI: 10.1109/IPDPS.2013.101.

[59] Joshua Lothian, Sarah Powers, Blair D Sullivan, Matthew Baker, Jonathan Schrock, and Stephen W Poole. “Synthetic graph generation for data-intensive HPC benchmarking: Background and framework”. In: *Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2013/339* (2013).

[60] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. "Graphlab: A new framework for parallel machine learning". In: *arXiv preprint arXiv:1408.2041* (2014).

[61] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[62] S. Madougou, A. L. Varbanescu, C. D. Laat, and R. V. Nieuwpoort. "A Tool for Bottleneck Analysis and Performance Prediction for GPU-Accelerated Applications". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 641–652. DOI: 10.1109/IPDPSW.2016.198.

[63] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[64] Frank J Massey Jr. "The Kolmogorov-Smirnov test for goodness of fit". In: *Journal of the American statistical Association* 46.253 (1951), pp. 68–78.

[65] Duane Merrill, Michael Garland, and Andrew Grimshaw. "Scalable GPU Graph Traversal". In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 117–128.

[66] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. "Scalable GPU graph traversal". In: *PPOPP 2012, New Orleans, LA, USA*. Feb. 2012, pp. 117–128.

[67] Paulius Micikevicius. "General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem." In: *PDPTA*. Vol. 4. 2004, pp. 1359–1365.

[68] Edward F. Moore. "The shortest path through a maze". In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292.

[69] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. "Data-driven versus Topology-driven Irregular Computations on GPUs". In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 463–474.

[70] NVIDIA. *CUDA C++ Programming Guide v10.0.130*. Sept. 2018. URL: https://docs.nvidia.com/cuda/archive/10.0/cuda-c-programming-guide/index.html#hardware-multithreading (visited on 18-1-2021). Archive link: https://web.archive.org/web/20210119192144/https://docs.nvidia.com/cuda/

archive/10.0/cuda-c-programming-guide/index.html#hardware-multithreading.

[71] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture. The World's Most Advanced Data Center GPU*. Tech. rep. Version WP-08608-001_v1.1. NVIDIA Corporation, Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 11-7-2018).

[72] Garson O'Toole. *In Theory There Is No Difference Between Theory and Practice, While In Practice There Is*. Apr. 14, 2018. URL: https://quoteinvestigator.com/2018/04/14/theory/ (visited on 7-5-2018). Archive link: https://web.archive.org/web/20180416010144/https://quoteinvestigator.com/2018/04/14/theory/.

[73] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[74] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[76] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[77] Sidney Redner. "How popular is your paper? An empirical study of the citation distribution". In: *The European Physical Journal B-Condensed Matter and Complex Systems* 4.2 (1998), pp. 131–134.

[78] Ryan A Rossi and Nesreen K Ahmed. "Networkrepository: A graph data repository with visual interactive analytics". In: *arXiv preprint arXiv:1410.3560* (2014).

[79] Arfon M Smith, Daniel S Katz, and Kyle E Niemeyer. "Software citation principles". In: *PeerJ Computer Science* 2 (2016). DOI: 10.7717/peerj-cs.86.

[80] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. "A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures". In: *IPDPS '13*. IEEE Computer Society, 2013.

[81] SQLite Development Team. *SQLite*. Version 3.7.17. May 20, 2013. URL: https://www.sqlite.org.

[82] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. "A hypercube-based encoding for evolving large-scale neural networks". In: *Artificial life* 15.2 (2009), pp. 185–212.

[83] Kenneth O Stanley and Risto Miikkulainen. "Efficient reinforcement learning through evolving neural network topologies". In: *Network (Phenotype)* 1.2 (1996), p. 3.

[84] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[85] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE. 2017, pp. 74–85.

[86] Walter F Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A Heinz. "Experimental evaluation in computer science: A quantitative study". In: *Journal of Systems and Software* 28.1 (1995), pp. 9–18.

[87] WF Tichy. "Should Computer Scientists Experiment More?" In: *Computer* 31.5 (1998), pp. 32–40.

[88] Yale University. *The Yale Literary Magazine*. v. 47. Herrick & Noyes, 1882. URL: https://books.google.nl/books?id=iJ9MAAAAMAAJ.

[89] Merijn Verstraaten. *Belewitte*. Version 1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6959684. URL: https://doi.org/10.5281/zenodo.6959684.

[90] Merijn Verstraaten. *Belewitte GPU Experiment Results*. Version v1.0.0. Aug. 2022. DOI: 10.5281/zenodo.6925023. URL: https://doi.org/10.5281/zenodo.6925023.

[91] Merijn Verstraaten. *Mix-and-Match Dataset*. Zenodo, Oct. 2018. DOI: 10.5281/zenodo.4317449. URL: https://doi.org/10.5281/zenodo.4317449.

[92] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. *Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach*. 2017. eprint: arXiv:1708.01159.

[93] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Mix-and-Match: A Model-driven Runtime Optimisation Strategy for BFS on GPUs". In: *Proceedings of the 8th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE. 2018, pp. 53–60.

[94] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Quantifying the Performance Impact of Graph Structure on Neighbour Iteration Strategies for PageRank". In: *"Euro-Par 2015: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2015, pp. 528–540. ISBN: "978-3-319-27308-2".

[95] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. "Synthetic Graph Generation for Systematic Exploration of Graph Structural Properties". In: *"Euro-Par 2016: Parallel Processing Workshops"*. Springer, Cham. "Springer International Publishing", 2016, pp. 557–570. ISBN: "978-3-319-58943-5".

[96] Vasily Volkov. "Understanding Latency Hiding on GPUs". In: (2016).

[97] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. "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*. ACM. 2016, p. 11.

[98] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific data* 3 (2016).

[99] G. Wu, J.L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. "GPGPU performance and power estimation using machine learning". In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. Feb. 2015, pp. 564–576. DOI: 10.1109/HPCA.2015.7056063.

[100] Marvin V Zelkowitz and Dolores R. Wallace. "Experimental models for validating technology". In: *Computer* 31.5 (1998), pp. 23–31.

[101] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. "Performance and Power Analysis of ATI GPU: A Statistical Approach". In: *NAS'11*. IEEE Computer Society, 2011.

[102] J. Zhong and B. He. "Medusa: Simplified Graph Processing on GPUs". In: *17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP'12* (2012), pp. 283–284.

[103] Konrad Zuse. *Der Plankalkül*. 63. Gesellschaft für Mathematik und Datenverarbeitung, 1972.

Glossary

ACM Association for Computing Machinery 18

AoS Array of Structures 54

API Application Programming Interface 34

ASCI Advanced School for Computing and Imaging 210

BBB Binaire Beslissingsboom, in het Engels: Binary Decision Tree (BDT) 195, 196

BC Betweenness Centrality 5, 133, 192

BDT Binary Decision Tree iv, 7, 100–109, 111–115, 119, 121, 127, 130–133, 140, 149, 150, 191, 192, 209

BEZ Breedte-eerst Zoekopdracht, in het Engels: Breadth-First Search (BFS) 195

BFS Breadth-First Search iv, v, 4, 5, 7, 24, 31, 34, 38, 39, 42, 43, 46, 58–65, 85, 100, 102–109, 111–120, 122–124, 126, 128, 140, 144, 190, 191, 209

BSP Bulk Synchronous Parallel 14, 33, 36–38, 132, 133

BVE Beeldverwerkingseenheid, in het Engels: Graphical Processing Unit (GPU) 193–196

CART Classification And Regression Trees 101

CDF Cumulative Distribution Function 77

CNN Convolutional Neural Network 101

CPPN Compositional Pattern Producing Network 76

CPU Central Processing Unit 2, 11, 13, 14, 31, 54

CSA Computer Systems Architecture viii

CSR Compressed Sparse Row 6, 51, 65, 105

CSV Comma-Separated Values 21–23

CUDA Compute Unified Device Architecture 11, 14, 17, 27, 43, 44, 56, 62, 88, 151

DAS5 Distributed ASCI Supercomputer 5 35, 56, 62

DFS Depth-First Search 5

DOI Digital Object Identifier 27

DSL Domain Specific Language 14, 42

EDF Empirical Distribution Function 77

FAIR Findability, Accessibility, Interoperability, and Reusability 20, 21, 27

FLOPS floating point operations per second 2, 125

GAS Gather-Apply-Scatter 15, 58

GPGPU General Processing on GPU ix, 2, 6, 7, 9, 11–14, 17–19, 22, 23, 33, 41, 42, 111, 125, 127, 128, 130, 131, 189

GPLv3 General Public License version 3 21

GPMC General Purpose Multi-Core 2

GPU Graphical Processing Unit viii, ix, 1–4, 6, 7, 9–15, 23, 26, 27, 31, 34, 37–39, 42–44, 51, 54, 56, 59, 62, 82, 84, 86, 93–96, 98–100, 105–109, 111, 112, 119, 121, 126, 129–133, 138, 189–191, 209, 210

HPC High-Performance Computing 2, 3, 5, 10, 41, 43, 125, 127, 189

irregular algorithm algorithms whose access patterns and branching behaviour depends on their input 2, 3, 41, 127, 189

ISA Instruction Set Architecture 14

KS Kolmogorov-Smirnov v, 77, 78, 80

NEAT NeuroEvolution of Augmenting Topologies 70–72

NUMA Non-Uniform Memory Access 13

NWO Nederlandse Organisatie voor Wetenschappelijk Onderzoek 20

OpenCL Open Computing Language 11, 14

PRNG Pseudo Random Number Generator 36, 102, 148

PTX Parallel Thread Execution 14, 88, 89, 91, 97, 151

RDM Research Data Management 19, 21

RSE Relative Standard Error 47, 48, 106, 107

SCC Single-chip Cloud Computer vii

SDK Software Development Kit 27

SIMD Single Instruction, Multiple Data 54

SIMT Single Instruction, Multiple Threads 12, 111, 130, 131

SM Streaming Multiprocessor 12–15, 59

SMT Simultaneous Multi-Threading 11, 12, 15

SNE System and Network Engineering viii

SoA Structure of Arrays 54

SoC System-on-Chip 10

SP Stream Processor 11–13, 111

SQL Structured Query Language 23, 141, 143

SSSP Single-Source Shortest Path 5, 133, 192

SVM Support Vector Machine 101

UTC Coordinated Universal Time 143

UvA Universiteit van Amsterdam vii, viii

VU Vrije Universiteit vii