Towards Feature-Aware Graph Processing on the GPU

by

Lynus Vaz

M.Sc. (Tech.), Birla Institute of Technology and Science, Pilani, 2009

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Lynus Vaz 2021
SIMON FRASER UNIVERSITY
Spring 2021

Copyright in this work rests with the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Lynus Vaz**
Degree:	**Master of Science**
Thesis title:	**Towards Feature-Aware Graph Processing on the GPU**
Committee:	**Chair:** Alaa Alameldeen Associate Professor, Computing Science
	Keval Vora Supervisor Assistant Professor, Computing Science
	Arrvindh Shriraman Committee Member Associate Professor, Computing Science
	William N. Sumner Committee Member Associate Professor, Computing Science
	Tianzheng Wang Examiner Assistant Professor, Computing Science

Abstract

Unlike traditional graph processing applications, graph-based learning algorithms like Belief Propagation and Multimodal Learning require complex data such as feature vectors and matrices residing on graph vertices and edges, and employ vector/matrix operations on this data. GPU-based high-performance graph processing frameworks utilize clever techniques to mitigate the effect of random global memory accesses arising from irregular graph structure, and also perform efficient load balancing. However, these frameworks are oblivious to algorithm-specific details like the nature of operations involved and the vertex/edge property types used, hence generating unnecessary random global memory accesses. Moreover, traditional graph processing frameworks constrain the user to follow a strict sequence of operations, ignoring the nuances of different control flows in graph-based learning algorithms.

In this thesis, we present Onyx, a *feature-aware* framework for graph-based learning algorithms on the GPU. Onyx employs a feature-aware processing model where each vertex property is collectively computed by a group of threads. This allows accesses to be coalesced into fewer global memory transactions, improving memory utilization. Onyx also incorporates dynamic vertex activation to perform sparse computations as vertex properties stabilize over time. The user expresses computations in the form of parallel operations on vertex and edge features, providing flexibility for custom control flows that support different kinds of graph-based learning algorithms.

To extract high performance, Onyx automatically folds multiple parallel vertex- and edge-feature operations into a single kernel at compile-time. This eliminates the overhead of repeated kernel launches, and permits the use of low-latency shared memory as intermediate storage. We utilize GPU instructions to efficiently perform collaborative operations across vertex and edge features such as normalization, reduction and feature-level change detection. Finally, as feature-aware processing reduces the computation done per thread, we organized the critical path in Onyx as pipelined steps to minimize expensive dependency stalls.

Our evaluation shows Onyx's feature-aware processing decreases atomic transactions while increasing global load efficiency. Together with change-driven computation this results in up to 20.3× speedup. We also implemented the graph-based learning algorithms on state-of-the-art GPU graph frameworks, and observe that Onyx outperforms them by up to 51.2×.

Keywords: Graph Processing; GPGPU; GPU Graph Analytics

Acknowledgements

First and foremost, I am grateful to my advisor, Prof. Keval Vora, for his patient advice and guidance. I thank my lab mates, Joanna Che, Kasra Jamshidi, Matthew Morrisson, Miao Liu, Mugilan Mariappan, Nachiketa Ramesh, Pourya Vaziri and Rakesh Mahadasa for their practical suggestions and kind words. Appreciation is due to Prof. Ramprasad Joshi, Durgesh Samant, Gowri Thampi and Sidhartha Agrawal for their unwavering support. Finally, I would like to thank my family; their understanding and constant encouragement kept me going.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been increasing interest in efficient graph analytics, as well as GPGPU computing. GPUs are capable of a higher instruction throughput and memory bandwidth than a similarly-priced CPU [2]. Therefore, several high-performance graph processing frameworks [34, 22, 14, 10, 11, 5, 20, 32, 8, 26, 35] employ the GPU's massively parallel capabilities to accelerate graph computations. These solutions develop clever techniques to reduce the impact of the graph structure irregularity on the symmetric GPU architecture for traditional graph computations like PageRank, shortest paths, connected components, etc. Recent works like [27] point out several interesting graph-based computations including graph-based learning algorithms like Graphical Model Inference [9] and Collaborative Filtering [37], that present new challenges due to the complexities involved.

Similar to traditional graph computations, graph-based learning algorithms like Belief Propagation [9], Alternating Least Squares [37] and Multi-Manifold Ranking [33] compute vertex properties iteratively in super-steps, and in each super-step, vertex properties are computed based on the values of neighboring vertices and adjacent edges. However, these algorithms require complex data residing on graph vertices and edges, and perform complex operations during each super-step which are uncommon in traditional graph processing algorithms. For instance, Alternating Least Squares performs matrix inverse and matrix-vector operations on vertices and edges, and Multi-Manifold Ranking [33] performs Hadamard inverse and Hadamard vector product. Existing GPU-based graph processing systems like [10, 28, 22, 34, 14] do not capture the nuances in graph-based learning algorithms, and hence are limited in terms of the performance they deliver and the programmability they offer.

Performance

Existing graph processing frameworks are designed to be oblivious to the algorithm-specific details like the nature of operations involved and the type of properties on vertices and edges. As a result, GPU threads in such frameworks end up computing each vertex or edge value in isolation. While this design delivers efficiency in traditional graph algorithms, it does not take advantage of the GPU's processing capabilities for complex operations.

Therefore, these frameworks, when used for graph-based learning algorithms, increase the GPU global memory transactions (both, reads and atomic writes) and underutilize global memory bandwidth. While graph processing typically involves a high number of random memory accesses due to the irregular graph structure (well known in prior research), such additional inefficiencies resulting from global memory transactions further exacerbates the overall performance for graph-based learning algorithms.

Programmability

Existing graph processing frameworks provide strict programming models that restrict the number of vertex and edge operations that can be invoked in each iteration, along with the order in which these operations get invoked. Graph-based learning algorithms, however, often have multiple sub-operations to be performed on vertices and edges, hence demanding custom algorithm-specific control flow for operations over vertices and edges. Moreover, expressing operations in a change-driven manner to leverage incremental processing further adds its own complexity. Thus, shoehorning graph-based learning algorithms into existing frameworks requires carefully considering each operation in the algorithm, which makes the job of expressing complex algorithms more difficult, and sometimes even impossible for certain cases.

1.1 Contributions

In this thesis, we develop Onyx, a feature-aware graph processing system. We expose the feature information from vertex and edge properties to the underlying runtime, which allows Onyx to treat features as first class elements across all of its modules.

1.1.1 Efficient Feature-Aware Processing

Onyx employs the compact CSR (Compressed Sparse Row) representation [25] to maximize graph size that can reside in the GPU global memory, and uses AOS (Array of Structures) format for vertex and edge properties to enable easier programmability. With this layout, Onyx maps threads to computations in a feature-aware manner. Specifically, each vertex (and edge) property is collectively computed using a group of threads such that consecutive threads operate on consecutive features of vertex (and edge) properties; since these features reside next to each other in global memory, their accesses get coalesced leading to efficient global memory bandwidth utilization. For efficient processing, the runtime eliminates redundant computations by incorporating *vertex activation*, where vertices and their outgoing edges get skipped if their property values do not change. To minimize warp divergence arising from the dynamically changing set of active vertices and edges, Onyx performs dynamic load balancing in feature-aware manner.

1.1.2 Flexible Feature-Guided Programming

Onyx employs a flexible programming model that does not enforce a strict control flow, and computations are expressed as parallel operations over features of vertex or edge properties; we call these operations *Feature-Guided Computations*. Programmers express their custom control flow in the form of ordering over the feature-guided computations. This allows writing different kinds of complex algorithms in form of the (mathematical) sub-steps involved, making it intuitive for domain experts. Furthermore, by focusing on individual sub-steps one at time, each operation can be easily expressed in *change-driven* fashion where changes in incoming feature values are used to directly change the outgoing feature values, hence enabling incremental processing.

1.1.3 Retaining High Performance

Developing feature-aware processing along with flexible feature-guided programming (as described above) poses several challenges from the performance standpoint, which we overcome using the following key techniques.

Kernel Folding to Minimize Operation Overheads

As complex algorithms get expressed in form of multiple sub-steps, the overheads from repeatedly rebalancing the same set of active vertex and edge properties can slow down the overall processing. To avoid this, we develop *kernel folding*, a technique that automatically combines (or *folds*) multiple sub-steps together (at compile time) so that their execution does not lead to repeated iteration and load balancing work. The folded kernels also allow utilizing shared memory across multiple operations to achieve higher efficiency, which would not be possible with separate kernel executions.

Efficient Feature-Aware Collaborative Operations

Since our algorithms rely on complex operations, we develop feature-aware collaborative operations in Onyx (e.g., normalization, reduction, feature-level change detection, standard matrix/vector operations), where a group of threads operating on different features of the properties cooperatively perform the operation. Our collaborative operations use GPU hardware instructions (e.g., `ballot` and `shfl`) and efficient strategies like tree-based reduction.

Pipelining for Instruction-Level Parallelism (ILP)

Decomposing the edge computations at the feature-level leads to dependency stalls since the edge function does not provide sufficient instruction-level parallelism (ILP) to hide the memory and execution latencies incurred during load balancing. We increase the ILP by unrolling one of our key loops and by dividing the sub-operations in pipelined steps.

1.2 Results

Our evaluation shows that Onyx's feature-aware processing decreases the number of atomic transactions and simultaneously increases global load efficiency; together with change-driven computation, this results in up to 20.3× faster execution. Furthermore, we implemented our graph-based learning algorithms on state-of-the-art graph processing frameworks including Tigr [22], SIMD-X [14] and Gunrock [34], and observed that Onyx outperforms them by up to 17×, 6.8× and 51.2× respectively.

1.3 Outline

This dissertation is organized as follows. Chapter 2 gives an overview of graph-based learning algorithms and motivates the need for feature-aware design. Chapter 3 discusses existing works on GPU-based graph processing. This is followed by a description of Onyx's programming model in Chapter 4 and processing model in Chapter 5 together with the optimizations. Chapter 6 describes Onyx's method of folding kernels. Finally, Chapter 7 presents the evaluation of Onyx to study its performance, and Chapter 8 concludes this dissertation.

Chapter 2

Background and Motivation

In this chapter, we will first briefly discuss GPU architecture and the CUDA programming model, and then describe the graph-based learning algorithms that we consider in this work. Finally, we will motivate the need for an efficient solution to handle graph-based learning algorithms by observing the limitations of existing graph frameworks.

2.1 GPU Architecture and Programming Model

We provide an overview of the GPU architecture and the CUDA programming model.

2.1.1 GPU Architecture

The GPU comprises an array of streaming multiprocessors (SMs) that can execute many threads concurrently depending on the hardware capabilities of the GPU, with 2048 being a typical limit. The SM uses the SIMT (Single Instruction Multiple Threads) model to handle this large number of threads. In this model, the SM manages and schedules threads in groups of 32 called warps that execute the same instruction in lock-step. Each thread of a warp is called a lane. The warp scheduler selects a warp to execute from those that are ready to run. As soon as an executing warp is blocked on memory, execution or some other dependency it is scheduled out and another is scheduled in. Branching is performed by executing each path while deactivating threads not on that path. This results in inactive threads uselessly participating in warp execution. Thus, *thread divergence* – having threads of a warp take divergent paths – reduces efficiency.

2.1.2 CUDA Programming Model

The CUDA programming model [2] is heterogeneous and allows a program running on a *host* (CPU) to launch multi-threaded code on a *device* (GPU). In this model, the host and device have separate memory spaces. This requires the use of APIs that enable the host to manage memory allocations in the device space and data transfer.

CUDA extends C++ with *kernels*, i.e., functions that are executed by many threads in parallel instead of just once. The CUDA threads that execute kernels are organized into thread blocks, each with an identifier of upto 3 dimensions. Thus, a kernel is launched as a *grid* of threads in 1, 2 or 3 dimensions. A thread can be identified within its block with a *local identifier* and within the grid by its *global identifier* that is a combination of the block identifier and its local identifier. This allows natural indexing into various kinds of data, including areas and volumes. All threads of a block are executed on the same processor core and share the limited resources of that core. Threads of the same block can synchronize between themselves and share memory.

When a kernel is invoked by a CUDA program on the host, the blocks of the grid are distributed to SMs that have available capacity. Each thread block is executed on a single SM and all its threads share the hardware resources of that SM. These resources include registers partitioned between the warps and shared memory that is partitioned between blocks. An SM may execute multiple thread blocks concurrently, subject to SM resources. As blocks terminate and free up resources, other blocks can be launched on the SMs that now have resources available.

2.1.3 Memory Model

CUDA threads are permitted to access a hierarchy of memory. First, each thread has its own local memory that is private. Next, each thread block provides shared memory that can be accessed by all the threads in the block and has the same lifetime as the block. Global memory can be accessed by all threads.

The GPU memory model maps naturally to the CUDA memory hierarchy. Global memory is present on the device, and can be accessed by all threads. The host program manages allocations in this memory space, and can copy data into and out of this space. On-chip memory has higher bandwidth and lower access times than global memory, making it a natural fit for scratchpad memory. This memory is shared by the threads in a block and is referred to as *shared memory*. Thread-local memory is implemented in GPU registers. Apart from these read-write memory areas, the GPU provides read-only *constant memory* that is served from the *constant cache*. Hits in this cache are served at a throughput similar to register accesses, while cache misses are serviced at a throughput of device memory. This memory is used to pass kernel arguments and can also be used for limited read-only data. Global memory is of the order of gigabytes, while shared memory and constant memory are much more limited. Shared memory varies from 48-100KB per SM depending on the GPU model, and constant memory is fixed at 64KB for the entire device respectively.

Global memory is accessed at a 32-byte granularity that corresponds to the GPU L2 cache line size. The GPU hardware coalesces global memory accesses by warp threads into memory transactions of the cache line size. Therefore, memory accesses that are correctly aligned and fully coalesced result in maximum efficiency since they use a minimum number of transactions.

If, however, scattered memory locations are accessed, the number of transactions increases leading to a lower effective bandwidth.

Atomic instructions are useful in serializing operations across threads to provide consistent results. The GPU provides instructions to perform atomic addition (`atomicAdd()`), subtraction (`atomicSub()`), compare-and-swap (`atomicCAS()`) and boolean operations (`atomicAnd()`, `atomicOr()`, `atomicXor()`).

CUDA provides several annotations to enable the programmer to tailor the application to the GPU architecture. A function marked with the `__global__` keyword is compiled into a kernel that can be launched with multiple threads on the GPU. `__device__` and `__host__` functions are compiled into device (GPU) and host (CPU) code respectively. Variables annotated with the `__shared__` keyword are allocated in shared memory, while the `__constant__` keyword specifies that a variable is to be allocated in constant memory.

2.1.4 Profiler Metrics

The GPU vendor provides profiling tools to guide programmers in obtaining high performance. The `nvprof` tool is used to collect metrics from the GPU to provide insight into the behavior and performance of a CUDA application. The *global load efficiency* metric is the ratio of requested global memory load throughput to required global memory load throughput expressed as a percentage [2]. A global load efficiency of 100% means that the required load throughput is the same as the requested throughput, i.e., only requested bytes are loaded from global memory. The *atomic transactions* metric is a count of the number of memory transactions generated by atomic operations. Atomic operations, too, follow coalescing rules for global memory. Therefore, a coalesced access pattern during atomic operations results in fewer atomic transactions performed than a scattered access pattern. Hence this metric can be used as a measure for memory coalescing during atomic writes.

2.1.5 Warp-Centric Programming

The GPU's SIMT architecture leads naturally to the *warp-centric* programming paradigm [7] that uses the warp as a cooperative unit. This is further promoted by the use of efficient hardware instructions that allow information sharing between warp threads. The `__shfl()` hardware instruction and its variants allow a thread to read a register from another thread in the same warp. The `__ballot()` instruction provides a bitmask of a boolean condition for each thread in the warp while the `__any()` instruction returns true if a boolean condition is true for any thread of the warp. The caveat here is that threads can only read data from other active threads in the warp participating in the exchange.

The massive number of GPU threads can be leveraged to compute aggregates, for instance, a sum, using a tree-based reduction. Instead of one thread reducing the inputs to a single value, a number of threads cooperatively reduce the value using a tree structure. The data can be exchanged using the `__shfl()` instructions described above (for intra-warp

reduction), or using shared memory or global memory (for reduction using more threads). This method reduces the time of the aggregation operations from $O(n)$ to $O(\log n)$, and has been used in many works [11, 14, 34] to improve performance.

2.2 Graph-based Learning: Details & Observations

While there has been plenty of research in developing efficient support for graph processing applications [7, 5, 10, 28, 4, 11, 22, 34, 14, 20, 21], none of them focus on the graph-based learning algorithms that we consider in this work. Hence we briefly discuss the core computation in graph-based learning algorithms, and contrast them with the traditional graph processing applications.

2.2.1 Definitions & Notations

Graph-based learning algorithms primarily compute values residing on vertices, and in some cases also residing on edges. We call these *vertex properties* and *edge properties*, and refer to their values as *property values*. For example, in PageRank algorithm (a standard graph processing benchmark) shown in Figure 2.1f, $pr(v)$ is the property of vertex v while $pr^i(v)$ is the value of pr property for vertex v at iteration i. Most of the properties in our algorithms are vector types, and each element within the vector is called a *feature*. Apart from vertex/edge properties, there may be constants for each edge or vertex (e.g., edge weight, scaling factor, etc.) whose values do not change throughout the execution; we call them *weights* and we use $w(u, v)$ to denote the weight of edge u to v. We use $N^-(v)$ and $N^+(v)$ to indicate the set of incoming and outgoing neighbors of vertex v. Throughout the paper, we use k to denote the number of features per vertex or edge property.

2.2.2 Graph-based Learning Algorithms

Figure 2.1 shows the key computation in several graph-based learning algorithms: Belief Propagation [9] (BP), Collaborative Filtering using Alternating Least Squares [37] (ALS), Multimodal Learning [30] (MML), Label Propagation [38] (LP), Multi-Manifold Ranking [33] (MMR). We also show PageRank [23] (PR) in Figure 2.1f which is a standard graph processing benchmark in order to contrast the different complexities involved in graph-based learning algorithms.

The algorithms progress iteratively in Bulk Synchronous Parallel (BSP) [31] manner, where property values in a given iteration get computed only after all the property values for the previous iteration get fully computed. Hence, in each iteration (superscripted with i), vertex property values (marked with red squiggles) get computed based on property values from previous iteration (superscripted with $i-1$) residing on the neighboring vertices. Operations like $\sum$ and $\bigcirc$ perform aggregation of property values coming from neighboring vertices along with edge weights, and hence, they translate into parallel edge operations

$$\overline{m^i_{uv}} = \left(\overline{b^{i-1}(u)} \circ (\overline{m^{i-1}_{vu}})^{\circ -1}\right) \cdot \Psi(u,v)$$
$$\overline{aggr(v)} = \bigcirc_{u \in N^-(v)} \overline{m^i_{uv}}$$
$$\overline{b^i(v)} = normalize(\overline{\phi(v)} \circ \overline{aggr(v)})$$

(a) Belief Propagation (BP)

$$\overline{sc(v)} = \sum_{u \in N^-(v)} \left(w(u,v) \cdot \overline{lf^{i-1}(u)}\right)$$
$$IC(v) = \sum_{u \in N^-(v)} \left(\overline{lf^{i-1}(u)}^T \cdot \overline{lf^{i-1}(u)}\right)$$
$$\overline{lf^i(v)} = \left(IC(v) + \lambda \bar{I}\right)^{-1} \cdot \overline{sc(v)}$$

(b) Alternating Least Squares (ALS)

$$\overline{waggr(v)} = \sum_{u \in N^-(v)} \overline{w(u,v)} \circ \overline{mm^{i-1}(u)}$$
$$\overline{haggr(v)} = \sum_{u \in N^-(v)} \overline{w(u,v)})^{\circ -1}$$
$$\overline{mm^i(v)} = \alpha.\overline{waggr(v)} \circ (\overline{haggr(v)} + (1-\alpha).\overline{mm^0(v)}$$

(c) Multi-Manifold Ranking (MMR)

$$\forall p \leq P:$$
$$\overline{aggr_p(v)} = \sum_{u \in N^-(v)} \left(w_p(u,v) \cdot \overline{lp^{i-1}(u)_p}\right)$$
$$\overline{n\_aggr_p(v)} = normalize(\overline{aggr_p})$$
$$\overline{prob_p(v)} = \alpha \cdot \overline{n\_aggr_p} + (1-\alpha) \cdot \overline{lp^{i-1}(v)_p}$$
$$\overline{lp^i(v)_p} = normalize(\overline{prob_p(v)})$$

(d) Label Propagation (LP)

$$AGGR(v) = \sum_{u \in N^-(v)} (\overline{w(u,v)}^T \cdot \overline{mmode^{i-1}(u)})$$
$$\overline{mmode^i(v)} = \bar{\alpha} \cdot AGGR(v) + (1 - \bar{\alpha} \odot \bar{1}).\overline{mmode^0(v)}$$

(e) Multimodal Learning (MML)

$$aggr(v) = \sum_{u \in N^-(v)} \frac{pr^{i-1}(u)}{|N^+(u)|}$$
$$pr^i(v) = \frac{1-\alpha}{|V|} + \alpha \cdot aggr(v)$$

(f) PageRank (PR)

Figure 2.1: Core computation in Graph-based Learning Algorithms (a-e), and in PageRank (f) for comparison (shown in mathematical form for brevity). Vector types are represented using an overline (e.g., $\overline{sc(v)}$), while matrices are represented in uppercase (e.g., IC). The red squiggly underlines indicate the vertex properties to be computed. Operators $\odot$, $\circ$ and $^{\circ -1}$ indicate dot product of two row vectors, Hadamard (element-wise) product of two vectors/matrices, and Hadamard (element-wise) inverse of a vector.

(similar to '`Advance`' in Gunrock [34], a popular GPU-based graph processing framework). Other operations like addition, multiplication, transpose and inverse that are computed on intermediate property values for vertices translate into parallel vertex operations (similar to '`Compute`' in Gunrock).

Key Observations

We observe that graph-based learning algorithms involve several important details that are absent in traditional graph analytics applications. These details can be categorized into two main classes, as described below.

- Complex Data Types: The data types of vertex/edge properties and weights are often complex. For example, vertex properties are of vector types, and edge weights are a mix

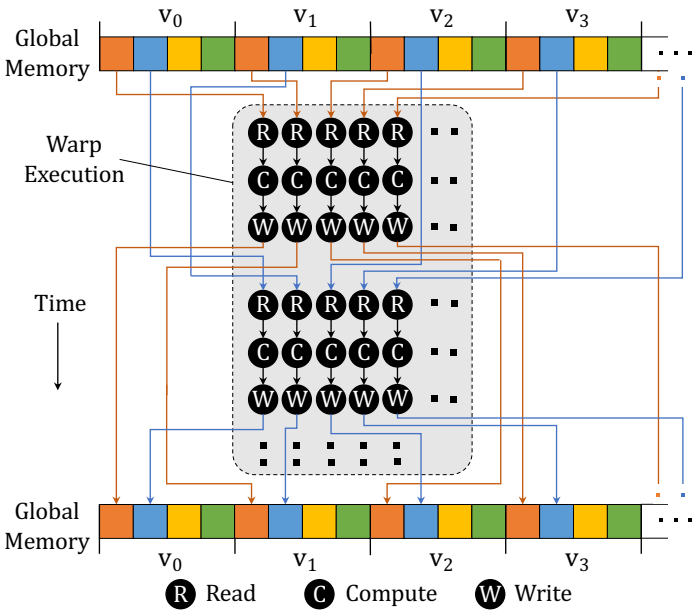


Figure 2.2: Example of an execution on traditional graph processing system, where each vertex property has 4 features (shown in different colors). Since traditional systems remain oblivious to the underlying vertex/edge property types, they generate spatially separated memory accesses resulting in an increase in memory transactions.

of scalars (ALS and LP), vector types (MML and MMR), and also matrix types (BP). In comparison, vertex properties in PageRank are scalars, and edges are unweighted [1].

- Complex Computations: The operations involved in these algorithms are complex as well; in fact, the computations often include multiple sub-steps, each of which are non-trivial. For example, they use operations like vector multiplication (BP), matrix inverse and matrix-vector operations (ALS), and Hadamard (element-wise) inverse (MMR). On the other hand, PageRank is computed using simple operations like scalar addition and multiplication.

2.3 Limitations of Existing Solutions

None of the existing graph processing solutions efficiently handle complex graph-based learning algorithms mainly because they remain oblivious to the underlying vertex/edge property types and the nature of computation involved. Furthermore, they often use strict programming models with fixed control flow that disallows easily expressing complex graph-based learning algorithms. Next, we discuss both of these problems in detail.

[1]Other traditional graph processing benchmarks like shortest paths have integer scalars as edge weights.

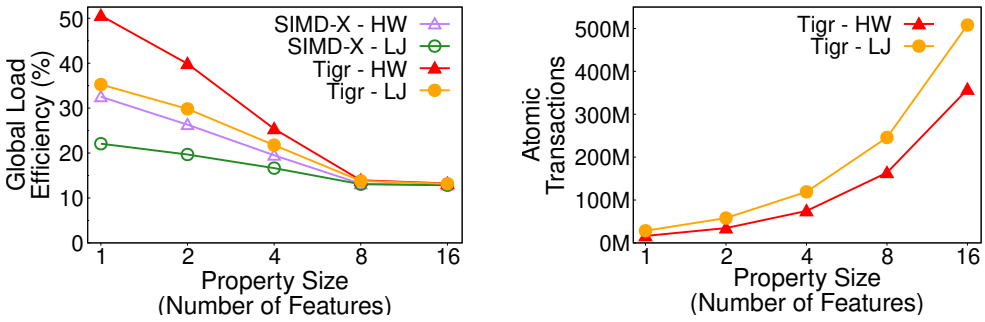


Figure 2.3: Performance of MMR on Tigr [22] and SIMD-X [14] as the number of features increase. SIMD-X uses pull mode which avoids atomic transactions, while Tigr's push mode incurs atomic transactions.

2.3.1 Performance

Since our graph-based learning algorithms primarily operate on non-scalar properties containing multiple features, performing vertex and edge computations by treating their properties as a single black box unit leads to inefficient usage of GPU global memory. As shown in Figure 2.2, when threads operate on consecutive edge or vertex properties, they process the individual features one-by-one. This causes consecutive threads to generate memory accesses that are spatially separated by k features. Since global memory read/write transactions are performed in 32-byte blocks, this spatial separation results in an increase in memory transactions that are scattered in different locations, resulting in lower effective memory bandwidth. To verify this behavior, we implemented and profiled the MMR algorithm (Figure 2.1c) on Tigr [22] and SIMD-X [14], two recent graph processing systems using the Hollywood2009 (HW) and SocLiveJournal (LJ) graph datasets (details available in Table 7.1). The results for both Tigr and SIMD-X, Figure 2.3 shows a decline in global load efficiency as properties become large, which means the amount of non-coalesced memory accesses increases as number of features increase. Furthermore, Tigr uses a push-based model with atomic operations which are expensive (SIMD-X avoids atomic operations using pull mode); as shown in Figure 2.3, the atomic transactions increase quickly with features per property.

Non-coalesced memory accesses decrease global memory throughput and performance [2]. While graph processing typically involves high number of random memory accesses due to the irregular graph structure (well known in prior research), such increase in inefficient memory accesses (both, reads and atomic writes as shown above) further exacerbates the overall performance of our graph-based learning algorithms.

Terminology	Explanation
Vertex/Edge Property	Value residing on a vertex/edge
Feature	An element of the vector property on each vertex/edge
Vertex Activation	Selectively processing vertices and/or their adjacent edges
Push Computation Mode	Vertex values are propagated over outgoing edges from a vertex (usually involves atomic writes to destination vertices)
Pull Computation Mode	Vertex values are propagated over incoming edges to a vertex (usually involves reading from source vertices)
Incremental Computation	Computing values by incrementally adjusting them based on new values, instead of recomputing the whole value
Change-Driven Processing	Propagating changes in values instead of entire values, and incrementally computing using the changes
Bulk Synchronous Parallel (BSP) model	Processing model that computes property values in iterations
Thread Block	A cooperative group of threads executing on the same GPU processor core
Warp	A group of 32 threads executing in lock-step on the GPU
Global Memory	On-device GPU memory accessible by all GPU threads
Shared Memory	On-chip GPU high-bandwidth memory accessible by block threads
Memory Transactions	Memory operations to read/write memory into/from GPU L2 cache
Memory Coalescing	Conversion of simultaneous warp thread memory accesses into multiple memory transactions by the GPU
Kernel	A function processed by threads in parallel on the GPU
Kernel Fusion	Combining multiple kernels into a single kernel

Table 2.1: A summary of terminologies used in this thesis.

2.3.2 Programmability

Since our graph-based learning algorithms involve multiple complex operations, the strict nature of existing programming models makes it difficult to express these algorithms efficiently.

To understand this issue, consider the ALS algorithm shown in Figure 2.1b. Each iteration involves two different aggregations across incoming edges of a vertex (sc and IC), and then a transformation to compute the final vertex property (lf^i). This is different from programming models used in traditional graph graph processing systems where each iteration involves a single operation over edges and a single operation over vertices. Furthermore, different algorithms perform different number of operations in each iteration: for example contrary to ALS, the LP algorithm shown in Figure 2.1d performs only a single aggregation over the edges, and then performs three transformation steps to compute the final vertex property (lp^i). Forcing all different algorithms to follow a strict programming model requires carefully examining each algorithm separately, and possibly rewriting certain operations to fit in the available functions/ callbacks. This makes expressing complex graph algorithms difficult, especially since the sub-operations involved (e.g., matrix inverse, normalization, etc.) are themselves not trivial.

Moreover, the strict programming model hinders development of efficient programs that could result in faster processing. For example, incremental processing is a popular technique to optimize graph-based computations, which is achieved by propagating *changes*

in property value across the edges, and updating the property values *incrementally* using the propagated changes [29, 19, 17, 16]. Such incremental processing requires expressing computations in *change-driven* fashion, which is not an easy task for complex algorithms like the ones considered in this work. A strict programming model only increases the difficulty to express such incremental computations.

The above limitations justify the need for a better design that can efficiently handle the nuances of graph-based learning algorithms.

2.4 Summary of Terminologies

Table 2.1 gives a summary of graph processing and GPU terminologies we use in this thesis for easy reference.

Chapter 3

Related Work

Several solutions have been proposed to efficiently process irregular graph computations on GPUs.

3.1 Graph Processing Systems

These works focus on traditional graph analytics benchmarks (e.g., PageRank, shortest paths, etc.), and address several important issues like irregular (non-coalesced) memory accesses, load imbalances, underutilization, kernel launch overheads, and many others. However, none of these works focus on the graph-based learning algorithms that we consider in our work, and hence, their processing models remain oblivious to the feature-level information for such algorithms. Moreover, several works utilize fixed programming models tailored for graph analytics benchmarks, making it difficult to express different control flows as required by our graph-based learning. Below we summarize the key contributions of these works.

[7] proposed the virtual warp programming method that balanced edge work across groups of warp threads (called virtual warps). CuSha [10] uses shards to represent the edges, so that edges in the shard get processed cooperatively by block threads which enables coalesced memory accesses. KiTES [11] uses the CSR representation [25] where warps cooperatively process edges incident on the vertices, and load balancing is achieved using a dynamic thread assignment strategy. Gunrock [34] proposes an Advance-Filter-Compute abstraction for graph analytics workloads. It implements scheduling strategies at CTA-, warp- and thread-level to load balancing, workload management and memory efficiency. Tigr [22] develops node-splitting transformations to reduce graph irregularity for load balancing, and enables coalesced memory accesses for its edge array. Totem [5] proposes a hybrid CPU-GPU graph processing engine, and explores partitioning strategies to allocate workloads matching to the strengths of the processing elements they get allocated to. Mapgraph [4] dynamically chooses among different scheduling strategies (CTA-, warp- and thread-level) depending on the frontier size and the size of the adjacency lists for vertices in the frontier. SIMD-X [14] utilizes an Active-Compute-Combine programming model, and develops different filtering

mechanisms to assemble the frontiers (active worklists). It also uses a push-pull based kernel fusion technique to reduce kernel call overheads. SEP-graph [32] develops a hybrid execution mode to switch between synchronous v/s asynchronous, push v/s pull and data-driven v/s topology-driven processing, and uses CTA scheduling, warp cooperation instructions, and priority scheduling for algorithms. GSwitch [20] provides a Filter-Expand processing model and uses a machine learning based algorithmic auto-tuner to dynamically choose the processing mode suitable for the graph algorithm.

Several works enable out of GPU memory and multi-GPU graph processing. Medusa [36] is a multi-GPU graph framework that uses the Edge-Message-Vertex processing model develops techniques including a cost model guided to reduce cross-GPU data transfer and overlapping computation with data transfer to hide transfer latencies. Merrill et al. [21] proposes a degree-dependent parallelization strategy with thread+warp+CTA scheduling. Lux [8] is a distributed multi-GPU system that exploits locality and aggregates memory bandwidth on GPUs. It provides push and pull execution modes, and proposes a dynamic graph repartitioning strategy to enable well-balanced workload distribution across GPUs. It provides an implementation of SGD algorithm that uses CTAs to load vertex features into shared memory, but the burden of extracting coalesced memory accesses using feature-level information is left on the user. Furthermore, it uses a single thread to process all the features of a given vertex and so loses the opportunity to cooperatively compute vector and matrix operations. GTS [12] maintains updateable attribute data in GPU memory and loads read-only graph topology data from the disk. It uses a slotted page format to handle differing vertex degrees, and overlaps computation and data transfer. GraphReduce [28] uses frontiers to transfer graph subsets containing active vertices/edges to the GPU, and provides a gather-apply-scatter programming abstraction. Graphie [6] proposes renaming algorithms for efficient shared memory usage and tracking partitions that should be copied to the GPU. Garaph [15] uses a replication scheme to handle differing vertex degrees, and utilizes a pull model for the GPU and a notify-pull model on the CPU. Subway [26] extracts active subgraphs from the CSR list to reduce the amount of graph data to be transferred to GPU. DiGraph [35] proposes a path-based asynchronous execution model that enables faster state propagation and a dependency-aware path dispatching on multi-GPUs.

Chapter 4

Programming Model

Inspired by the programming flexibility in Ligra [29], Onyx allows expressing graph computations as parallel operations over edges and vertices. Onyx also provides advanced programming support including: (a) an API to enable the automatic kernel folding (discussed later in Chapter 6); and, (b) a rich set of feature-aware collaborative implementations for standard operations like normalization, feature-level change detection, matrix/vector operations, etc. (discussed later in Chapter 5).

Figure 4.1 shows Onyx's key edge-parallel `edgemap()` and vertex-parallel `vertexmap()` API, and Figure 4.2 shows the ALS algorithm (from Figure 2.1b) expressed in Onyx. The `vertexmap` and `edgemap` (lines 43-45) invoke the user-defined functions on the *features* of 'active' vertex and edge properties respectively. The number of features k is captured statically via template argument to `vertexmap` and `edgemap`. The active vertices are identified using the `active` bit-vector whereas active edges are identified as outgoing edges of vertices marked in the `active` bit-vector. Both functions also return the set of vertices whose values get changed by marking them in the bit-vector (2nd argument on line 45). This allows dynamically operating on the changing values by chaining `edgemaps` and `vertexmaps`.

Since Onyx does not enforce a strict control flow across `vertexmaps` and `edgemaps`, programmers can easily construct custom control flows for different algorithms. As shown in lines 43-45, the different steps of ALS are expressed as separate parallel operations over

```
1  <int k, typename udata>
2  void vertexmap(bitvec* active_in, bitvec* active_out,
                  graph<udata>* g,
                  bool vfunc(uint, uint, udata*));

3  <int k, typename udata>
4  void edgemap(bitvec* active_in, bitvec* active_out,
                graph<udata>* g,
                bool efunc(uint, uint, uint, uint, udata*));
```

Figure 4.1: Key APIs of Onyx.

```
5  template<int k>
6  __device__ bool compute_ic(uint fIdx, uint u, uint v,
                              uint edgeIdx, udata* d) {
7    __shared__ float m1[blockDim * k];
8    __shared__ float m2[blockDim * k];
9    float *my_lf_prev = d->lf_prev[u];
10   float *my_lf_curr = d->lf_curr[u];
11   float *my_M1 = &m1[(blockTID - fIdx) * k];
12   float *my_M2 = &m2[(blockTID - fIdx) * k];
13   vec_trans_mul<k>(fIdx, my_M1, my_lf_prev, my_lf_prev);
14   vec_trans_mul<k>(fIdx, my_M2, my_lf_curr, my_lf_curr);
15   float *my_IC = &d->IC[v];
16   for (int i = 0; i < k; i++) {
17     atomicAdd(&my_IC[i*k + fIdx],
                 my_M2[i*k + fIdx] - my_M1[i*k + fIdx]);
18   }
19   return true;
20 }

22 template<int k>
23 __device__ bool compute_sc(uint fIdx, uint u, uint v,
                              uint edgeIdx, udata* d) {
24   float rating = d->rating[edgeIdx];
25   float lf_change = d->lf_curr[u][fIdx] - d->lf_prev[u][fIdx];
26   atomicAdd(&mySC[fIdx], rating * lf_change);
27 }

29 template<int k>
30 __device__ bool compute_lf(uint fIdx, uint v, udata* d) {
31   /* Skipped: Compute my_lf_new[fIdx] using IC & sc */
32   d->lf_prev[v][fIdx] = d->lf_curr[v][fIdx];
33   d->lf_curr[v][fIdx] = my_lf_new[fIdx];
34   bool my_updated = (fabs(d->lf_curr[v][fIdx] -
                        d->lf_prev[v][fIdx]) > EPSILON);
35   return any_feature_updated<k>(fIdx, my_updated);
36 }

38 void main(void) {
39   /* Skipped: Load graph data on GPU */
40   bitvec active;
41   initialize(&active, true);  // Allocate and fill on GPU
42   do {
43     edgemap<4>(&active, NULL, &cf_data, d_compute_sc);
44     edgemap<4>(&active, NULL, &cf_data, d_compute_ic);
45     vertexmap<4>(NULL, &active, &cf_data, d_compute_lf);
46     sync_device_with_host();
47   } while (!is_converged(active));
48 }
```

Figure 4.2: ALS algorithm implemented on Onyx. The `d_compute_ic`, `d_compute_lf` and `d_compute_sc` are lambda wrappers to invoke `compute_ic`, `compute_lf` and `compute_sc` device functions respectively. Onyx calls are highlighted in color; the orange calls are Onyx's feature-aware collaborative operations (discussed in Section 5.3).

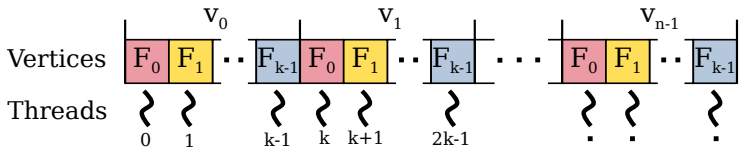


Figure 4.3: Vertex properties laid out in AOS format enabling coalesced memory accesses.

edges and vertices. While the two edge computations (`d_compute_ic` and `d_compute_sc`) with separate `edgemap` calls appear inefficient at first glance, in Chapter 6 we will see how they get combined together to retain high performance.

4.1 Expressing Feature-Guided Computations

To express the vertex/edge features to the Onyx runtime, we design a *feature-guided* programming model. Here, computations over edges and vertices are expressed at the feature-level. Hence, the feature values read from and written to the global memory get correctly addressed based on the properties being computed.

The array of structures (AOS) layout, shown in Figure 4.3, colocates features of each edge/vertex in a structure that is straightforward for the user. However, SIMT performance often benefits from rearranging data into separate arrays for each edge/vertex feature, which is called the structure of arrays (SOA) layout [1]. While SOA layout is recommended for efficiency, it requires the users to carefully map the computations correctly in order to enjoy its performance benefits. In comparison, the AOS layout places lesser burden on programming, and hence Onyx allows the user to use the intuitive AOS layout while retaining performance.

With the AOS layout, consecutive threads in a warp operate on different features of the same property (details about how Onyx maps feature-level computations to threads will be discussed in Chapter 5). The underlying runtime ensures that all the features of a given property get processed by threads within a single warp; this simplifies programming by eliminating the burden of enforcing inter-warp coordination. Furthermore, the runtime passes an index referring to the feature within the property (`uint fIdx`) as the first argument to the user-defined edge and vertex functions; this allows easy feature-level addressing. In our ALS program, `compute_ic()` reads the relevant information for the current feature (using `fIdx`) from `IC`, `lf_prev` and `lf_curr`, and then writes to the vector corresponding to the feature in the `IC` matrix (lines 15-18).

4.2 Expressing Change-Driven Computations

Onyx's vertex activation allows computations to be skipped on certain vertices and their outgoing edges, based on an input bit vector. This improves performance by not recomputing

vertices and edges that have already stabilized. However, computing over a dynamically changing set of edges can end up generating different results on BSP algorithms if performed naively, especially when the underlying runtime operates in push-mode (i.e., when values are pushed out by source to destination, instead of being pulled in by destination from the source). Hence, frameworks like Tigr [22] and SIMD-X [14] disable vertex activations for such algorithms and compute all graph edges in each iteration.

The key to correctly utilizing vertex activation is to perform *change-driven computations* as performed in CPU-based graph systems like [29, 17, 16], which enables capturing the right subset of edges on which computations get invoked. Hence, we use *change-driven programming* in Onyx where values are computed incrementally by incorporating the new changes in previously computed values. We use a similar strategy as PageRank-Delta from [29] where the aggregated values (i.e., result of *sum*, *product*, etc.) from incoming neighbors are saved on vertices, and then the changes in incoming neighbors' property values incrementally adjust the saved aggregated values. As shown in Figure 4.2, `compute_ic` incrementally adjusts `IC` using changes from `lf_prev` and `lf_curr`.

Due to the complexities in our algorithms, the change in value is often computed in two steps: first computing the old value and new value (e.g., `my_M1` and `my_M2` on lines 13-14), and then computing the difference between them using basic operators like add/subtract or multiply/divide (e.g., `atomicAdd` on line 17). For certain algorithms like Belief Propagation (BP shown in Figure 2.1a), the change itself cannot be directly expressed from the old and new values alone; for such cases, first the old value is incrementally removed from the aggregation value, and then the new value is incrementally added, resulting in two separate atomic operations.

After the final feature value gets incrementally recomputed in the `vertexmap`, it is compared with the previous value to identify whether any feature value changed significantly (line 34). Then, our feature-aware collaborative operation `any_feature_updated()` is used to identify changes across all features within the vertex property (line 35), based on which the runtime automatically activates the vertex in the bit-vector.

Chapter 5

Feature-Guided Processing Model

Onyx performs *feature-aware processing* which maps threads to vertex and edge features. We first show how this mapping is achieved efficiently in presence of change-driven computations, and then discuss the feature-aware collaborative operations provided by Onyx.

5.1 Thread Mapping & Dynamic Load Balancing

Figure 5.1 shows how feature-aware processing happens in `edgemap` and `vertexmap` functions.Each vertex and edge property is collectively computed by a group of parallel threads such that threads with consecutive identifiers operate on contiguous features (as shown in Figure 5.2). Since features of vertices/edges are stored in an AOS layout, this results in coalesced memory accesses. For features less than the warp size, this maps naturally to warp threads that execute in lock-step, and each warp may compute several edges or vertices simultaneously. For more features, an entire thread block is used to collectively process a single vertex or edge.

With change-driven computations, only active vertices (i.e., the ones whose properties change) and their edges are processed. The `active_in` bit-map is used to identify the active vertices and prune out computations resulting from unchanged properties. However, naively disabling computations on inactive edges leads to warp underutilization as some threads operate on active edges while other threads that get mapped to inactive edges perform `no-op`. To eliminate this underutilization, we use a load balancing strategy where computations for active edges get dynamically mapped to available warp threads, as described next.

5.1.1 Balancing Feature Computations for Edges

We take inspiration from load balancing strategies in works like [11] where threads get assigned to active edge computations only. Here, the prefix sum of the degrees of the active vertices assigned to the group (warp or block) are calculated. Then, the edges are distributed uniformly over the group of threads. Each thread computes the end points of the edge assigned to it using a binary search into the prefix sum array. Since this strategy does not

```
template<int k, typename udata>
void edgemap(bitvec* active_in, bitvec* active_out, graph<udata>* g, bool
    efunc(...)) {
  /* spawn g->n / warpSize warps on the GPU */
  for w in 0 to g->n / warpSize - 1 in parallel {
    __shared__ volatile uint scan[blockDim];
    uint vertex = globalTID;
    v_active = is_active(active_in, vertex);
    vertexDegree = v_active ? v_degree : 0;
    /* Write the exclusive prefix scan into scan[] array. scan[] becomes an
        index to the edge lists of vertices assigned to the warp.*/
    numEdges = prefix_scan(scan, w, vertexDegree);
    for (e = laneId; e < numEdges; e += warpSize) {
      /* Per-edge phase */
      (edgeIdx,edgeSrc,edgeDest) = binary_search(e,scan,w);
      /* Per-feature phase */
      for(lane=laneId/k; lane<warpSize; lane+=warpSize/k) {
        eIdx = __shfl(edgeIdx, lane);
        u = __shfl(edgeSrc,lane); v= __shfl(edgeDest,lane);
        int fIdx = laneId % k;
        ret = efunc(fIdx, u, v, eIdx, g->data);
        if (active_out) {
          uint warpUpdated = __ballot(ret);
          if (fIdx == 0 && (warpUpdated & kMask(laneId, k)))
              atomicOr(&active_out[v / warpSize], mask(v));
        }
      }
    }
  }
}

template<int k, typename udata>
void vertexmap(bitvec* active_in, bitvec* active_out, graph<udata>* g,
          bool vfunc(...)) {
  // spawn g->n / warpSize warps on GPU
  for warp w in 0 to g->n / warpSize - 1 in parallel {
    uint myVertex = globalTID; bool vUpdated = false;
    for (i = laneId / k; i < warpSize; i += warpSize / k) {
      uint v = w * warpSize + i;
      bool ret = false;
      if (is_active(active_in, v)) {
        int fIdx = laneId % k;
        ret = vfunc(fIdx, v, g->data);
      }
      warpUpdated = __ballot(ret);
      vUpdated=vUpdated || warpUpdated & vMask(laneId,i,k);
    }
    warpUpdated = __ballot(vUpdated);
    if (active_out && laneId == 0)
      active_out[myVertex / warpSize] = warpUpdated;
  }
}
```

Figure 5.1: Feature-aware EdgeMap and VertexMap.

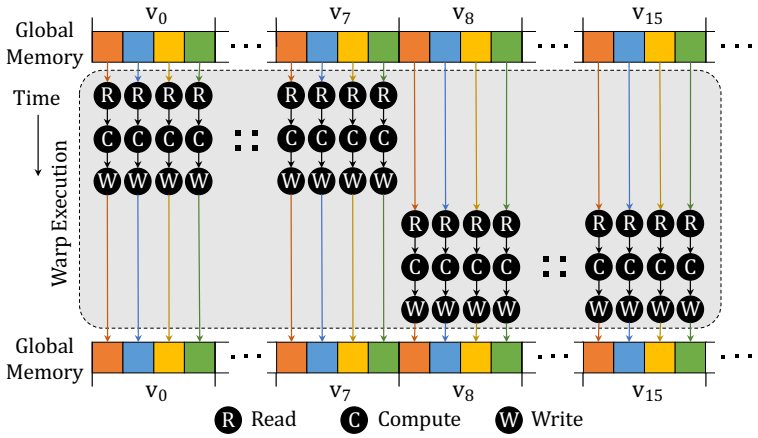


Figure 5.2: Memory accesses with feature-aware mapping (number of features = 4). Consecutive threads read and write consecutive feature values of vertices, causing the accesses to be coalesced. GPU coalesces memory at the quarter-warp level (i.e., 8 threads).

consider vertex properties as collections of features, we develop a feature-aware dynamic load balancing strategy in Onyx where warp threads get assigned to individual feature computations corresponding to active vertices and edges.

To map threads on features of active edges, we compute the exclusive prefix sum of the degree of each active vertex in the warp, and perform a binary search across the prefix sum array, after which threads iterate over individual features of the assigned edges. The binary search operation is a tight loop that cannot be pipelined since every iteration depends on the result of the previous one, which causes significant stalls due to execution dependencies. To avoid these expensive stalls, our feature-aware load balancing is performed in two phases: a *per-edge phase* followed by a *per-feature phase*.

During the per-edge phase (lines 60-61), the edge index and its corresponding source and target vertices get computed using a single binary search per edge. Then, in the per-feature phase (lines 62-73), these edges and their features are re-distributed cooperatively over the warp: each thread reads the results of the binary search from other threads that are computed in the per-edge phase (using `__shfl()` on lines 64-65). By doing so, the expensive binary search operation is performed only once for each active edge, and the final mapping results in consecutive threads assigned to consecutive features of each edge. We observed further stalls due to the binary search and reduced edge computation code, which we describe in Section 5.2.

If the feature size is greater than the warp size, all threads in a warp compute the features of the same vertex or edge simultaneously. Therefore, the warp threads do not diverge in such cases and so we switch off the dynamic thread assignment.

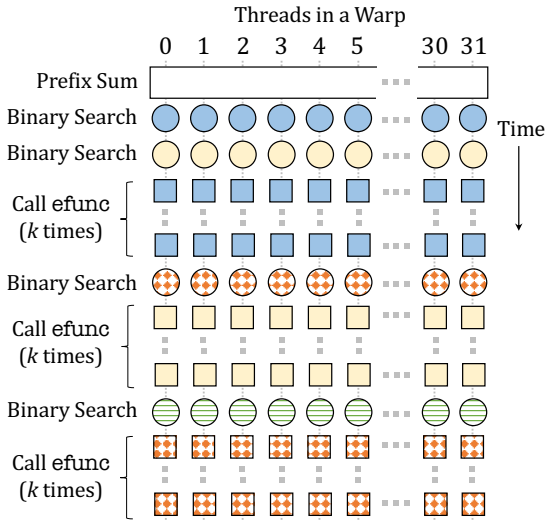


Figure 5.3: Pipelining the sub-operations in EdgeMap by unrolling the loop. The results of binary search are computed in the current iteration and used in the next iteration to avoid stalls.

5.1.2 Balancing Feature Computations for Vertices

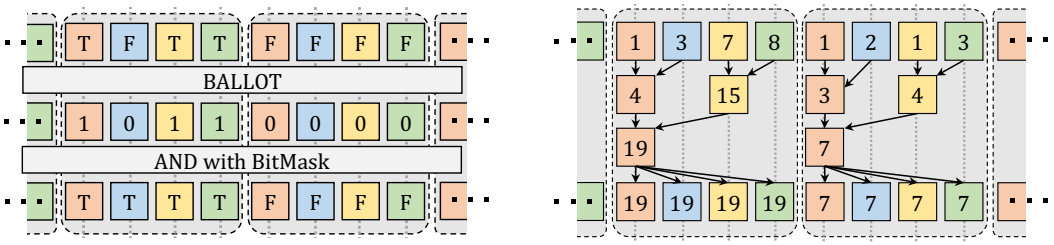
Feature-aware load balancing for computations on vertex features is simpler since they are invoked directly once the active vertex gets identified (i.e., no need to iterate over edges). As shown in lines 83-92, threads in each warp cooperatively process the features of active vertices, and each feature computation is directly performed by the thread responsible for it.

5.1.3 Capturing Active Vertices

For change-driven computation, the `active_out` bit-vector captures the active vertices as ones whose values get updated. This is determined based on the return value of user functions `efunc` and `vfunc`. Since these functions operate at feature-level, their return values get aggregated at edge-/vertex-level using the `__ballot()` hardware instruction (lines 69 and 90). For features greater than the warp size, we require an extra write into shared memory for the result of each warp. The active vertex is then marked directly in `vertexmap()`, whereas it is marked using `atomicOr()` hardware instruction in the `edgemap()` (line 71) to avoid concurrent overwrites from multiple edges.

5.2 Pipelining for Instruction-Level Parallelism (ILP)

By decomposing the edge computations at feature-level, the user-defined edge function becomes smaller. This leads to dependency stalls since the edge function does not provide sufficient ILP to hide the memory and execution latencies incurred during load balancing. Furthermore, due to aliasing constraints, the compiler is also unable to extract ILP by reordering code across global writes [18].



(a) Collaborative update detection (number of features = 4).

(b) Collaborative tree-based reduction (number of features = 4).

Figure 5.4: Feature-aware collaborative operations in Onyx.

To increase ILP, we manually unroll the edge loop (lines 59-74), as summarized in Figure 5.3. We divide the edge loop into steps: the first step performs the binary search to identify base addresses, and the second step performs the feature-level computation. These two steps are pipelined such that instructions for the second step (over a given edge) are interleaved with the instructions for the first step (over the next edge). By doing so, the code size increases which allows the compiler to improve ILP by optimizing across larger code.

5.3 Feature-Aware Collaborative Operations

Computing features for a given property is not always embarrassingly parallel, and it often requires reading other features in the same property. For example, the normalization operation (used in Label Propagation, Figure 2.1d) reads all the feature values in a given property and scales them to a well-defined range. For such computations, we develop feature-aware collaborative operations in Onyx, where a group of threads operating on different features of the same property cooperatively perform the operation.

Our collaborative operations include: normalization, reduction, and `any_feature_updated()` (used in Figure 4.2 on line 35) which identifies whether the vertex should be active based on changes in its features.

These operations are implemented using warp-level hardware instructions like `__shfl_up()`, `__shfl_down()` and `__ballot()` to retain maximum efficiency (summarized in Figure 5.4c-d). As described in Section 2.1.1, these hardware instructions allow threads in the same warp to share data. Thus, threads read feature information from other threads simultaneously when processing the same vertex/edge. When processing features greater than the warp size, we require temporary storage in order to exchange data between threads. For this we allocate a user-defined size of on-chip shared memory, which avoids the more expensive global memory accesses. In order to increase efficiency, our implementation uses warp-level instructions wherever possible before synchronizing the entire thread block. The implementations of these operations serve as a template to develop custom collaborative computations.

Onyx also provides collaborative feature-aware implementations for traditional operations like matrix-vector multiplication and matrix inverse that are performed on property vectors and matrices (useful for various algorithms, as seen in Figure 2.1).

Chapter 6

Kernel Folding for Performance

While our `vertexmap` and `edgemap` abstractions allow easily expressing complex graph algorithms in form of the sub-steps involved, executing the sub-steps separately one-by-one would cause repeated work of iterating through and rebalancing the same set of active vertex and edge properties. We instead automatically combine multiple sub-steps together into a single kernel. Thus, our form of kernel fusion, that we term kernel folding, retains easier expressivity of complex computations in the form of sub-steps while still extracting high performance at runtime. Furthermore, our kernel folding technique allows advanced programmers to express computations in an optimized manner, that would have not been possible with separate kernels. We first define the semantics of kernel folding to understand the properties that are guaranteed by the resulting folded kernels, and then present how folded kernels get generated at compile time.

6.1 Kernel Folding Semantics

We introduce the `foldmap()` abstraction to express the operations that must be folded into a single kernel. `foldmap` accepts a variable number of device functions, which can be a mix of vertex functions and edge functions. Figure 6.1 shows two functions from our ALS example (see Figure 4.2) getting invoked using `foldmap`.
The execution of folded kernels retains three key properties:

- Ordering: The edge and vertex functions within the folded kernel get executed in the same order as the order in which the user specifies them in `foldmap`. This allows easier development of complex algorithms using step-by-step (or ordered) operations.
- Progress: A given edge or vertex function within the folded kernel gets executed only if its predecessor function (i.e., the function within the folded kernel immediately before this function) returns `true`. Hence, if any function within the folded kernel returns `false` on a given vertex or edge, then the rest of the functions in the folded kernel do not get executed on that vertex or edge. This allows skipping computations that will not lead to any value changes.

```
98  foldmap<4>(&active,NULL,&cf_data,d_compute_sc,d_compute_ic);
```

Figure 6.1: Kernel folding in ALS program. Line 98 replaces lines 43-44 in Figure 4.2.

- Activation: Execution of the first function in the folded kernel on any given vertex/edge is determined by the input bit-vector, and the return value of the last function that gets executed is used to activate vertices in the output bit-vector.

6.1.1 Discussion

Kernel fusion improves performance by merging the vertex function with edge function. In frameworks such as [34] the fused vertex function is performed in the edge traversal sequence. This means that the vertex function may be performed multiple times requiring the user to ensure correctness by duplication-tolerant techniques like atomic instructions. [14] maintains barriers between the two functions, i.e., the vertex function gets called only after edge function has executed on all the edges. Onyx on the other hand aims to efficiently fuse multiple vertex and edge functions without synchronization using expensive barriers. Since the fused functions are not synchronized, the execution for vertices/edges in different warps may overlap each other. However, the ordering of these functions are retained at a single edge and vertex level.

6.2 Generating Folded Kernels

Kernel folding ensures that all the user-defined vertex/edge functions on a given vertex or edge feature get executed by the same GPU thread. To infer the user-defined function types (either edge function or vertex function), Onyx generates folded kernels at compile time (sample shown in Figure 6.2). During compilation, the user-defined functions get grouped into lists depending on their types (either vertex function or edge function) while still retaining the ordering specified in `foldmap`. This happens by repetitively finding the maximal continuous sub-sequence of same function type and putting those functions in a separate group. For each group, a call to `folded_edgemap()` or `folded_vertexmap()` is placed inside the folded kernel (lines 103-109) with the functions in that group statically passed in those calls.

As shown in Figure 6.3, the `folded_edgemap` function is similar to the `edgemap` function, except that it processes a list of edge functions. It statically captures (using variadic templates) the list of user-defined functions, and invokes them in order. It performs the feature-aware load balancing only once for the entire group which eliminates the redundant work, and then it invokes the edge functions one by one on active edges. `folded_vertexmap` similarly processes a list of vertex functions.

```
template<int k, typename udata>
void folded_kernel(bitvec* active_in, bitvec* active_out, VFunc vfunc1,
          VFunc vfunc2, ...EFunc efunc1, EFunc efunc2, ...EFunc efuncp) {
  // spawn g->n / warpSize warps on GPU
  for warp w in 0 to g->n / warpSize - 1 in parallel {
   /* folded edgemaps and vertexmaps in order */
   ...
   folded_vertexmap<k>(active_in,NULL, vfunc1, vfunc2, ...);
   folded_edgemap<k>(active_in, NULL, efunc1, efunc2, ...);
   ...
   /* active_out is only updated by last group */
   folded_edgemap<k>(active_in, active_out, efuncp, ...);
  }
}
```

Figure 6.2: Folded kernel generated at compile time.

```
template<int k, typename udata, typename... EFuncs>
__device__ void folded_edgemap(bitvec* active_in, bitvec* active_out,
      EFuncs... efuncs) {
  /* Feature-aware load balancing (lines 59-65 from Figure 5.1)*/
  int fIdx = laneId % k;
  for (each efunc in efuncs...) {
    ret = efunc(fIdx, u, v, eIdx, d_graph_data->data);
    if (!ret)
      break;
  }
  /* Update active_out (lines  68-72 from Figure 5.1) */
}
```

Figure 6.3: Folded variant of Feature-aware EdgeMap.

6.2.1 Shared Memory Caching via Kernel Folding

Since folded kernels process multiple user functions in a single kernel, threads within a GPU block remain alive throughout the execution of the folded kernel. This allows threads to utilize block shared memory and avoid global memory round-trips. For example, in ALS (Figure 2.1b), the lf^{i-1} vectors are loaded to compute sc and IC values in their respective kernels; when these kernels get folded into a single kernel, the lf^{i-1} vectors can be loaded in the shared memory to enable faster access for subsequent read requests. During runtime, the folded kernel allocates shared memory space at the beginning of its execution (before running the first `folded_edgemap()` or `folded_vertexmap()`), and releases the space at the end of its execution (after the last `folded_edgemap()` or `folded_vertexmap()` finishes).

Chapter 7

Experimental Evaluation

In this chapter, we evaluate Onyx to answer the following key questions.

1. How well does feature-aware processing perform with and without change-driven computations?
2. Do the benefits of feature-aware processing improve as the number of features increase?
3. How well do folded kernels bridge the performance gap between unfolded kernels and optimized kernels written by hand?
4. How does the performance of Onyx compare with traditional graph processing systems?

7.1 Methodology

To separate out the benefits of our feature-aware processing model from the effects of change-driven computation, we thoroughly evaluate the following variants of Onyx:

- **Onyx-NFA/NCD**: This is Onyx without feature-aware processing and change-driven computation (**NFA** = No Feature-Aware ; **NCD** = No Change-Driven). With both of these key features disabled, the approach becomes similar to existing graph processing frameworks.
- **Onyx-FA/NCD**: This is Onyx with feature-aware processing, but without change-driven computation (**FA** = Feature-Aware Enabled). This allows us to measure the effectiveness of feature-aware processing in isolation, without the benefits from change-driven computation.
- **Onyx-NFA/CD**: This is Onyx without feature-aware processing, but with change-driven computation (**CD** = Change-Driven Enabled). This allows us to measure the effectiveness of change-driven computation in isolation, without the benefits from feature-aware processing.

Graph	Vertices	Edges	Size (k = 4 / k = 8)
(CP) CitPatents	3.8M	16.5M	1.1 / 2.1 GB
(RC) RoadCentral	14.1M	16.9M	2.8 / 4.8 GB
(RU) RoadUSA	23.9M	28.9M	4.7 / 8.2 GB
(D24) DelaunayN24	16.8M	50.3M	3.9 / 7.0 GB
(HW) Hollywood2009	1.1M	56.4M	3.0 / 5.6 GB
(LJ) SocLiveJournal1	4.8M	68.5M	3.9 / 7.3 GB
(KN) KronG500L21	2.1M	91.0M	4.9 / 9.1 GB
(OK) SocOrkut	3.0M	106.3M	5.8 / 10.7 GB

Table 7.1: Input datasets (acquired from [13, 3, 24]). Since the memory footprint depends on the number of features (k) in the vertex and edge properties, the maximum size across all of our algorithms is reported for 4 and 8 features.

- **Onyx-FA/CD**: This is Onyx with feature-aware processing and change-driven computation.

Kernel folding is enabled across all of our experiments, and in Section 7.7 we will evaluate kernel folding by comparing its performance with the case when it is disabled.

7.2 Graph-based Learning Algorithms

We evaluate the performance of Onyx on the graph-based learning applications shown in Figure 2.1, and compare the results with the state-of-the-art high performance graph processing systems: **Tigr** [22], **Gunrock** [34] and **SIMD-X** [14]. Since none of these systems natively provide implementations for graph-based learning algorithms, we implemented each of our graph-based learning algorithms in them.

7.3 Graph Datasets

Table 7.1 lists the real-world graphs that we use in our evaluation. These graphs are a mix of social network graphs and road networks covering different densities. Since the graph-based learning applications primarily maintain vector of values on vertices and edges (as opposed to just a single scalar value), they have a much larger memory footprint compared to the traditional graph processing algorithms like PageRank. For example, the largest OK graph with 4-8 features takes 5.8-10.7GB space, which ends up occupying majority of the available GPU memory. Unless otherwise stated, our experiments use 4 features per vertex/edge; in Section 7.6 we will also study the performance as the number of features per vertex/edge changes.

7.4 Evaluation Platform

All experiments were conducted on Tesla P4 GPU with 7612MB GDDR5 device memory. The GPU is connected to a host machine (Google cloud N1 4 core, 96GB RAM) via PCIe 3x16

	Graph	Onyx			
		NFA/NCD	FA/NCD	NFA/CD	FA/CD
LP	CP	1170.6 ms	2.4×	3.4×	**7.3**×
	RC	2440.3 ms	1.5×	3.2×	**5.2**×
	RU	1954.9 ms	1.4×	1.9×	**3.1**×
	D24	2660.3 ms	1.4×	2.2×	**3.5**×
	HW	1989.7 ms	1.5×	6.7×	**10.1**×
	LJ	10349.6 ms	1.8×	9.3×	**16.4**×
	KN	2783.8 ms	1.2×	4.2×	**5.1**×
	OK	9026.9 ms	2.2×	5.0×	**10.4**×
ALS	CP	723.4 ms	2.6×	3.8×	**6.7**×
	RC	260.1 ms	1.4×	1.5×	**2.0**×
	RU	371.4 ms	1.2×	1.4×	**1.8**×
	D24	531.3 ms	1.7×	1.8×	**2.5**×
	HW	1189.9 ms	2.6×	5.6×	**12.3**×
	LJ	1644.5 ms	2.7×	3.6×	**9.0**×
	KN	4058.8 ms	1.5×	6.5×	**9.0**×
	OK	6407.6 ms	3.2×	7.6×	**20.3**×
BP	CP	517.5 ms	2.2×	3.0×	**4.2**×
	RC	196.5 ms	1.4×	1.2×	**1.8**×
	RU	349.5 ms	1.7×	1.1×	**1.9**×
	D24	572.1 ms	2.0×	1.4×	**2.2**×
	HW	857.6 ms	1.8×	2.0×	**2.5**×
	LJ	6496.3 ms	1.9×	1.1×	**2.7**×
	KN	2051.5 ms	1.7×	2.7×	**2.9**×
	OK	2244.8 ms	1.4×	2.4×	**2.9**×
MMR	CP	1331.3 ms	3.2×	3.2×	**8.5**×
	RC	2173.1 ms	2.0×	4.0×	**4.9**×
	RU	1213.2 ms	1.2×	1.7×	**2.1**×
	D24	1114.6 ms	1.2×	1.3×	**1.6**×
	HW	921.0 ms	1.3×	4.2×	**5.9**×
	LJ	8468.5 ms	2.8×	6.2×	**14.2**×
	KN	3806.1 ms	2.3×	3.0×	**6.6**×
	OK	10106.5 ms	2.9×	4.4×	**13.1**×
MML	CP	1057.7 ms	2.9×	3.6×	**8.4**×
	RC	685.7 ms	1.6×	3.3×	**4.4**×
	RU	692.4 ms	1.3×	2.3×	**2.8**×
	D24	878.2 ms	1.3×	2.3×	**3.0**×
	HW	1151.6 ms	1.7×	3.8×	**6.3**×
	LJ	4159.6 ms	2.0×	3.8×	**8.0**×
	KN	3609.4 ms	2.2×	3.5×	**6.0**×
	OK	6876.2 ms	3.2×	3.4×	**8.7**×

Table 7.2: Performance of Onyx. Absolute numbers for execution times are shown for Onyx-NFA/NCD, which forms the baseline for all other relative numbers. Speedups in bold text are the highest among the different variants.

link with 15.754 GB/s. The system ran 64-bit Ubuntu 18.10 and programs were compiled using the CUDA 10.1 toolchain with -O3 optimization enabled. GPU metrics were collected using the `nvprof` tool in the CUDA toolkit.

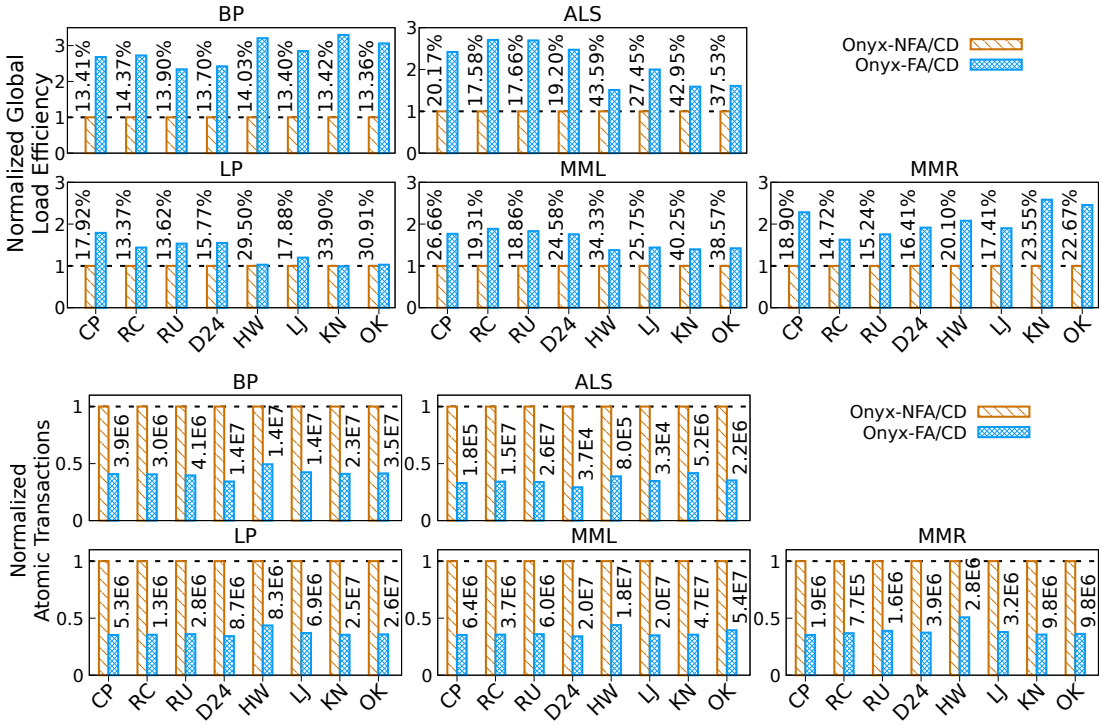


Figure 7.1: Global load efficiency and number of atomic transactions for Onyx-FA/CD normalized w.r.t. Onyx-NFA/CD. Absolute numbers are shown for atomic transactions on Onyx-FA/CD, and for global load efficiency on Onyx-NFA/CD.

7.5 Performance of Feature-Aware Processing

Table 7.2 shows the performance of Onyx's feature-aware processing. Comparing Onyx-FA/NCD v/s Onyx-NFA/NCD shows the benefits of feature-aware processing when change-driven computation is disabled, whereas comparing Onyx-FA/CD v/s Onyx-NFA/CD shows the benefits of feature-aware processing with change-driven computation.

7.5.1 Feature-Awareness with Change-Driven Computation

Onyx-FA/CD improves the overall performance by 1.6-20.3× compared to Onyx-NFA/NCD. The change-driven computation is effective in eliminating redundant computation for vertex and edge values that do not change, and hence it alone speeds up the processing by 1.1-9.3×. Feature-aware processing further boosts the performance by 1.1-3.0×, as seen by comparing Onyx-FA/CD v/s Onyx-NFA/CD.

To reason about the performance improvements, we studied how memory gets accessed during both load and store operations through GPU metrics provided by the profiling tool `nvprof`. Since atomic write operations heavily influence the overall performance, we measured the amount of *Atomic Transactions* performed throughout the execution. While the total number of atomic updates invoked by the application remains exactly the same,

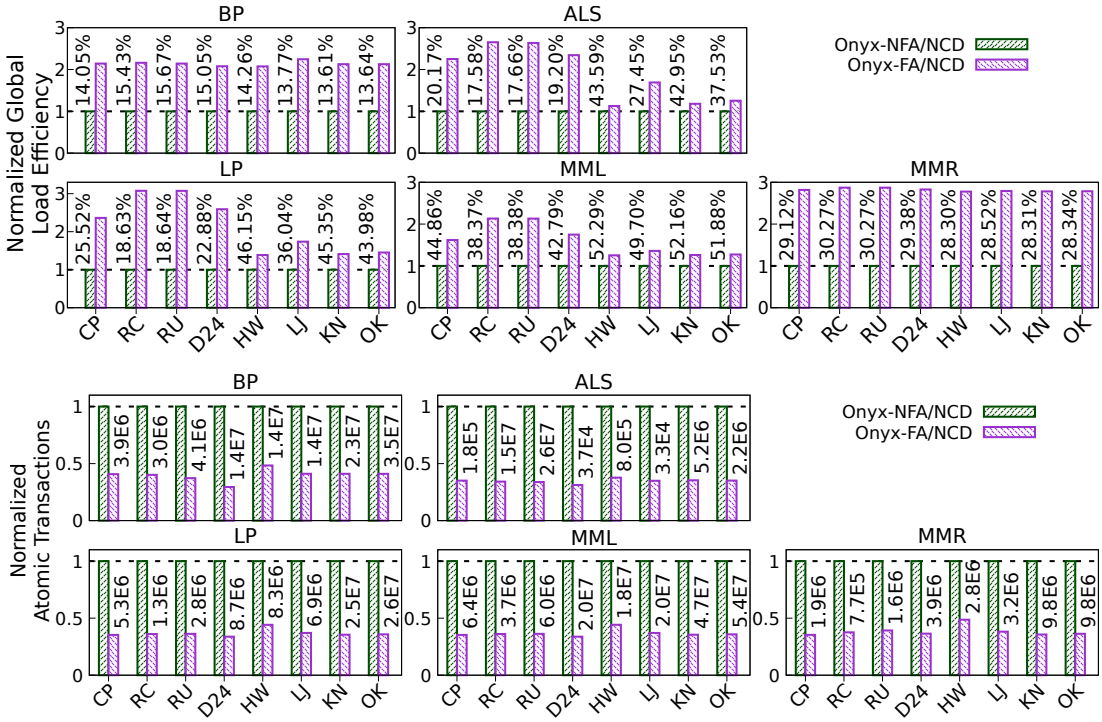


Figure 7.2: Global load efficiency and number of atomic transactions for Onyx-FA/NCD normalized w.r.t. Onyx-NFA/NCD. Absolute numbers are shown for atomic transactions on Onyx-FA/NCD, and for global load efficiency on Onyx-NFA/NCD.

the number of atomic transactions is expected to vary depending on the nature of memory accesses; for example, coalesced memory writes would result in fewer atomic transactions (which is favorable), whereas non-coalesced (or irregular) writes would cause more atomic transactions (which is expensive). As shown in Figure 7.1, the number of atomic transactions reduces by 2.0-3.4× with feature-aware processing, mainly because of its improved memory coalescing while updating the vertex feature values.

We also measured the *Global Load Efficiency* GPU metric, which is defined as the ratio of requested global memory load throughput to required global memory load throughput expressed as percentage [2]. This metric captures how efficiently the global memory bandwidth is utilized during load operations (higher value is better). As shown in Figure 7.1, the global load efficiency increases by 1.0-3.2× with feature-aware processing, again due to fewer memory transfers resulting from coalesced memory accesses. The amount of coalescing achieved during execution is mainly dependent on the graph algorithm and the input graph structure.

7.5.2 Feature-Awareness without Change-Driven Computation

For executions without change-driven computations, feature-aware processing achieves 1.2-3.2× speedup, as observed in Table 7.2 by comparing Onyx-FA/NCD v/s Onyx-NFA/NCD.

Figure 7.2 shows the number of atomic write transactions and the global load efficiency ratio for Onyx-FA/NCD normalized w.r.t. Onyx-NFA/NCD; again, the atomic write transactions decrease by 2.4-3.4× while load efficiency increases by 1.1-3.0× with feature-aware processing, mainly due to coalesced memory accesses.

7.5.3 Performance Summary

Both of our techniques, feature-aware processing and change-driven computation, individually contribute to the performance improvements. Since these techniques are orthogonal to each other (optimizing for memory accesses v/s optimizing for computation), they complement each other well: together, these techniques result in 1.6-20.3× performance improvement.

7.6 Sensitivity to Number of Features

The number of features in the graph algorithm directly impacts its performance, simply because the amount of computation increases with the number of features: for example, an algorithm with 8 features will roughly perform twice the amount of computation and memory accesses compared to the same algorithm with 4 features (assuming all other dynamic characteristics like vertex stabilization are same across the two executions). Since our feature-aware processing is aimed to improve performance in presence of features, we study its performance as number of features varies.

Figure 7.3 shows the performance of Onyx-FA/CD v/s Onyx-NFA/CD and Onyx-FA/NCD v/s Onyx-NFA/NCD when the number of features increases from 1 to 128. As expected, the execution times increase with increase in features; however, the increase in execution times with feature-aware processing is gradual (as shown with Onyx-FA/CD and Onyx-FA/NCD) whereas the execution times without feature-aware processing increase rapidly. This trend is mainly due to the improved coalescing of memory accesses with feature-aware processing; as seen in Figure 7.3 the increase in atomic transactions follows a similar gradual trend for feature-aware processing as opposed to a faster increasing trend for processing without feature-awareness. Overall, our feature-aware processing with 128 features ends up performing up to 7.7× faster compared to the execution without feature-awareness.

7.7 Effectiveness of Kernel Folding

We study how automatic kernel folding improves performance by comparing the execution times of our folded kernels with the unfolded kernels in Figure 7.4. We also show the performance of the hand-optimized variant for each case where the kernels are manually folded and optimized to achieve the best performance. The results for BP and LP algorithms are shown in order to cover folding of both, edge and vertex functions (edge functions get

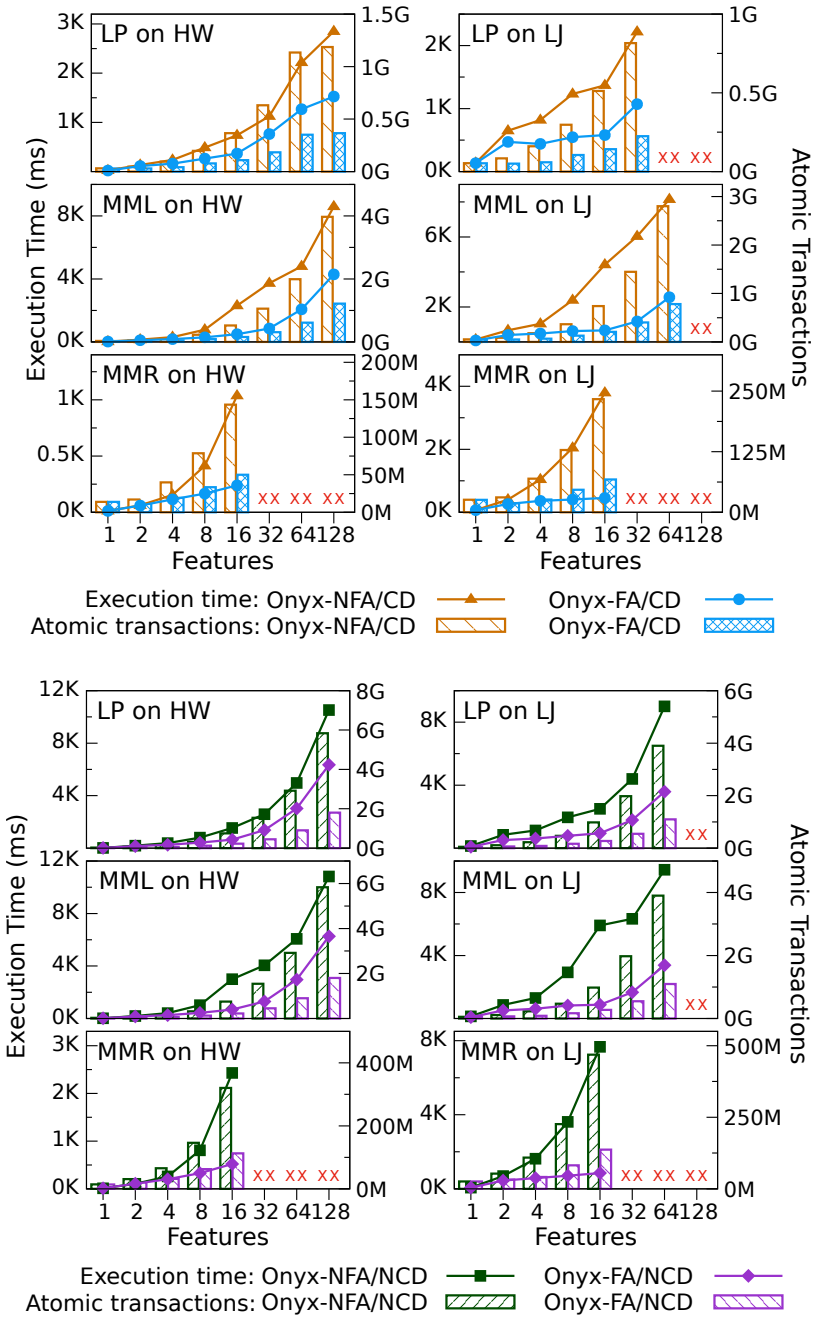


Figure 7.3: Execution times in milliseconds (on left y-axis) and number of atomic transactions (on right y-axis) for varying number of features. The X indicates executions that ran out of GPU memory.

folded in BP, while vertex functions get folded in LP). Both of our techniques, feature-aware processing and change-driven computation, are enabled in this experiment.

As seen in Figure 7.4, folded kernels perform 1.2-1.7× faster than unfolded kernels. This is mainly because folded kernels eliminate repeated iteration and load balancing work,

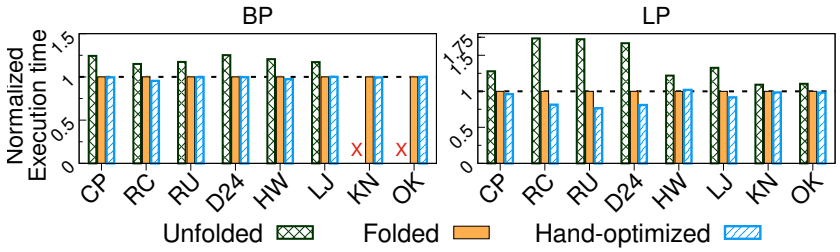


Figure 7.4: Performance of kernel folding by comparing execution times for 3 variants: (a) Unfolded = without kernel folding; (b) Folded = with kernel folding; and, (c) Hand-optimized = with manual folding and optimizing. The X indicates BP with unfolded kernels ran out of memory since it has to rely on global memory for intermediate data.

while also enabling shared memory caching across multiple operations. Furthermore, shared memory caching eliminates the need to use global memory for intermediate results, and hence, folded kernels have a smaller memory footprint compared to the execution with unfolded kernels; this is why BP with unfolded kernels could not run on KN and OK graphs, whereas it ran successfully with folded kernels. Finally, the performance of folded kernels is close to the hand-optimized variants for BP mainly because both of these variants perform the same amount of global and shared memory accesses. On the other hand, the hand-optimized variants perform faster for LP; this is because the hand-optimized variants reuse intermediate results in registers, whereas our automatically folded kernels utilize shared memory for intermediate results, which ends up limiting the number of thread blocks that can run in parallel in each GPU SM.

7.8 Feature-Aware v/s Traditional Graph Processing

We compare the performance of Onyx with three state-of-the-art traditional graph processing systems: Gunrock [34], Tigr [22] and SIMD-X [14].

None of these systems provide implementations for graph-based learning algorithms; hence, we implemented the algorithms on each of the system for this evaluation. In doing so, for each system we selected the processing mode that gave the best performance for that system. Gunrock provides vertex activations, hence we selected push mode with change-driven computation for Gunrock. Tigr does not provide vertex activations and implements algorithms in push mode, so we used push mode without change-driven processing on Tigr. SIMD-X provides vertex activations for certain algorithms such as shortest paths. However, its callback implementations are not suitable for BSP algorithms, which are implemented in pull mode without vertex activations. Hence, we implemented our algorithms on SIMD-X in the same style: pull mode without change-driven computations. Also, none of the systems provide an easy way for users to allocate temporary memory as scratch space across different functions. Hence, as part of our best effort to report their best performance without modifying the underlying system, we were able to use shared memory for Tigr and global

System	Computation Mode	Change-Driven Processing	Shared Memory
Gunrock	Push	Y	N
Tigr	Push	N	Y
SIMD-X	Pull	N	N

Table 7.3: Summary of implementation choices for Gunrock, Tigr and SIMD-X systems.

	Graph	Gunrock	Tigr	SIMD-X	Onyx
LP	**CP**	1345.4	1160.9	394.6	**159.6**
	RC	5700.7	2632.2	2412.2	**472.3**
	RU	7677.7	2106.7	3223.6	**630.7**
	D24	8282.2	3206.1	3860.5	**767.9**
	HW	741.8	1566.2	895.6	**197.6**
	LJ	5085.0	10754.1	3979.0	**630.2**
	KN	3449.6	5232.3	1131.5	**542.1**
	OK	5884.8	8406.8	2671.7	**865.4**
ALS	**CP**	5515.3	509.3	447.0	**107.8**
	RC	5941.4	423.2	391.8	**132.2**
	RU	X	675.1	697.6	**205.7**
	D24	X	781.4	1252.5	**214.8**
	HW	3122.4	585.7	471.9	**96.6**
	LJ	6633.0	1029.9	729.3	**182.8**
	KN	13199.5	6449.0	1511.5	**448.8**
	OK	10550.3	3414.8	1711.8	**315.8**
BP	**CP**	433.6	470.9	775.2	**122.9**
	RC	451.9	232.1	415.4	**111.1**
	RU	X	506.1	820.4	**186.5**
	D24	X	617.9	1709.0	**275.8**
	HW	570.7	903.5	1206.7	**349.0**
	LJ	X	9807.1	11324.6	**2425.5**
	KN	X	1687.7	X	**700.8**
	OK	X	3463.8	X	**761.0**
MMR	**CP**	537.7	1509.8	445.3	**156.7**
	RC	927.7	2594.7	828.8	**447.2**
	RU	1289.6	1548.6	1064.5	**582.9**
	D24	1620.1	2357.8	1976.7	**694.8**
	HW	245.9	833.4	1063.5	**155.4**
	LJ	1697.4	8715.8	3819.6	**594.8**
	KN	1816.3	5113.4	1512.6	**579.0**
	OK	2610.1	9745.6	3377.7	**773.1**
MML	**CP**	832.0	903.5	229.0	**126.2**
	RC	597.1	720.2	327.6	**157.3**
	RU	987.6	741.4	560.4	**244.4**
	D24	1041.1	966.5	641.6	**288.1**
	HW	403.9	728.1	316.5	**183.8**
	LJ	2807.1	3723.4	1143.9	**521.3**
	KN	3704.1	3463.8	634.3	**598.2**
	OK	5062.2	7335.8	1139.6	**792.6**

Table 7.4: Execution times (in milliseconds) for Onyx, Gunrock, Tigr and SIMD-X. An X indicates the execution ran out of memory.

memory for Gunrock and SIMD-X. Table 7.3 summarizes the implementation choices for each of the systems we compare against.

As shown in Table 7.4, Onyx outperforms all the other systems mainly because of its feature-aware processing model. Tigr's node split transformations reduces load imbalance while SIMD-X's pull-based processing eliminates the expensive atomic writes, which allows these systems to outperform Gunrock. Nevertheless, Onyx is up to 17× faster than Tigr and up to 6.8× faster than SIMD-X because Onyx achieves higher memory coalescing for both read and write transactions: the global load efficiency for Onyx is up to 5.6× higher than Tigr and 5.7× higher than SIMD-X, and Onyx performs 2.3-3.8× lesser atomic transactions compared to Tigr.

Chapter 8

Conclusions & Future Directions

8.1 Conclusion

We presented Onyx, a feature-aware graph processing system for complex graph-based learning algorithms. Onyx provides a feature-aware processing model where each vertex property gets collectively computed by a group of threads. This coalesces the memory accesses generated and reduces the memory transactions (both, atomic writes and reads). To further improve performance Onyx utilizes vertex activations along with change-driven computations that avoid recomputing vertices after convergence. To reduce the potential performance loss from warp divergence, Onyx performs dynamic load balancing that reduces warp divergence. For programmability, we designed a feature-guided programming model that provides the user full flexibility in defining custom control flows. This enables expressing complex graph-based learning algorithms directly in terms of the mathematical steps involved. In order to eliminate repeated load-balancing stages, we develop a kernel folding technique that combines multiple vertex and edge functions into a single kernel that preserves the properties of ordering, progress and activation. For efficiency, Onyx also provides feature-aware collaborative operations that employ GPU hardware instructions and cooperative tree-based strategies. Further, we pipelined load-balancing stages which hides the latency by increasing the instruction-level parallelism.

Our evaluation of Onyx showed that feature-aware processing delivers a speedup of up to 3.2×. When combined with change-driven processing, we observed a speedup of up to 20.3×. Onyx's kernel folding strategy generates folded kernels that perform 1.7× faster than regular kernels. Finally, Onyx is significantly faster than state-of-the-art graph processing frameworks: up to 51.2× over Gunrock, up to 17× over Tigr, and up to 6.8× over SIMD-X.

8.2 Future Directions

As part of future work, we plan to scale out Onyx to enable feature-aware processing across a multi-GPU setup in order to harness the processing power of multiple GPUs and scale

to larger datasets. Consider a cluster of machines, each with multiple GPUs connected by NVlink interconnects, and fast RDMA between machines. Processing graph-based learning algorithms on such a setup will involve transferring k times more data over the network compared to traditional graph processing algorithms, which can significantly limit the performance gains of utilizing the computation power of multiple GPUs.

The challenges here include partitioning data between machines to reduce network communication, and also synchronization between machines.

- The decision of whether to replicate data on each machine or stream from a 'master' must be based on the network bandwidth as well as how often the data is updated. In addition, since each machine has multiple GPUs, the interconnect bandwidth will decide whether data can be shared between such GPUs.

- BSP algorithms require synchronization between super-steps. However, instead of synchronizing at every super-step, partial aggregates can be calculated, and then synchronized at intervals. This comes at the cost of memory capacity, which is a limited resource on GPU chips.

Each of these factors needs to be considered in the framework design, and a tradeoff must be evaluated for performance.

Bibliography

[1] Nathan Bell and Jared Hoberock. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359–371. Morgan Kaufmann, Boston, MA, USA, 2012.

[2] NVIDIA Corporation. CUDA C++ Programming Guide. November 2019.

[3] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1), December 2011.

[4] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 345–354, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques, PACT '17*, pages 233–245, 2017.

[7] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 267–276, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, November 2017.

[9] U Kang, Duen Horng, and Christos Faloutsos. Inference of Beliefs on Billion-Scale Graphs. In *The Second Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[10] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium*

on High-Performance Parallel and Distributed Computing, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Enabling Work-Efficiency for High Performance Vertex-Centric Graph Analytics on GPUs. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A Fast and Scalable Graph Processing Method Based on Streaming Topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 447–461, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. 2015.

[14] Hang Liu and H. Howie Huang. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 411–428, USA, July 2019. USENIX Association.

[15] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient GPU-Accelerated Graph Processing on a Single Machine with Balanced Replication. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, page 195–207, USA, 2017. USENIX Association.

[16] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '21)*, pages 1–16, 2021.

[17] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth European Conference on Computer Systems*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Swapna Matwankar. CUDA 7.5: Pinpoint Performance Problems with Instruction-Level Profiling. September 2015.

[19] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research*, CIDR '13. www.cidrdb.org, 2013.

[20] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 201–213, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 117–128, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 622–636, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[24] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

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

[26] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing Data Transfer during out-of-GPU-Memory Graph Processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, 2017.

[28] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: Processing Large-Scale Graphs on Accelerator-Based Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Hanghang Tong, Jingrui He, Mingjing Li, Changshui Zhang, and Wei-Ying Ma. Graph Based Multi-Modality Learning. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, page 862–871, New York, NY, USA, 2005. Association for Computing Machinery.

[31] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[32] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 38–52, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Yang Wang, Muhammad Aamir Cheema, Xuemin Lin, and Qing Zhang. Multi-Manifold Ranking: Using Multiple Features for Better Image Retrieval. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 449–460, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[34] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and et al. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing*, 4(1), August 2017.

[35] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: An Efficient Path-Based Iterative Directed Graph Processing System on Multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 601–614, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Jianlong Zhong and Bingsheng He. Medusa: A Parallel Graph Processing System on Graphics Processors. *SIGMOD Record*, 43(2):35–40, December 2014.

[37] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, AAIM '08, page 337–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] Xiaojin Zhu and Zoubin Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. In *CMU Technical Report CALD-02-107*, 2002.