Universal Graph Framework: Achieving High-Performance across Algorithms, Graph Types, and Architectures

by

Ajay Rajendra Brahmakshatriya

Bachelor of Technology, Indian Institute of Technology Hyderabad (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2020

Certified by .....
Professor Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Universal Graph Framework: Achieving High-Performance across Algorithms, Graph Types, and Architectures

by

Ajay Rajendra Brahmakshatriya

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The performance of graph programs depends highly on the algorithm, the size and structure of the input graph, and the features of the underlying hardware. No single set of optimizations or single hardware platform works well across all applications. Currently, when switching to a different hardware platform, programmers must re-implement graph algorithms in a completely different language or framework, and use different optimizations to achieve high performance.

We propose the Universal Graph Framework (UGF), a new graph processing framework that achieves high performance across CPUs, GPUs, and Domain-Specific Accelerators (DSAs) automatically, using the same algorithm specification. UGF achieves portability with reasonable effort by decoupling algorithm, schedule, and backend. We introduce a new domain-specific intermediate representation, GraphIR, that is key to this decoupling. GraphIR encodes high-level algorithm and optimization information needed for hardware-specific code generation, making it easy to develop different backends (GraphVMs) for diverse architectures spanning CPUs, GPUs, and DSAs. UGF builds on the GraphIt domain-specific language (DSL), over which it introduces a new extensible scheduling language that separates hardware-independent and hardware-specific transformations. The scheduling language enables combining load balancing, edge traversal direction, active vertex set creation, kernel fusion, and other optimizations on GPUs and can be extended to support other hardware backends, such as CPUs and DSAs. We also built an autotuner on top of UGF to automatically find the best schedules for different hardware platforms.

We demonstrate that UGF's techniques enable high performance and portability across a wide range of architectures by building three backends that target highly diverse hardware platforms: GPUs, CPUs, and the Swarm DSA. We evaluate UGF on five algorithms and 9 input graphs on these architectures. UGF outperforms state-of-the-art frameworks by up to 5.1×, and is the fastest in 62 out of 90 experiments.

Thesis Supervisor: Professor Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

To Mom, Dad and Jai

Acknowledgments

I would like to express my gratitude to many people without whom this thesis would not have been possible.

I would like to thank my advisor, Professor Saman Amarasinghe, for believing in me and guiding me for two years. Saman always asked me to think big and work on problems that have real impact and patiently supported me till I found one. I have learnt a lot from Saman's problem before solution approach and hope to continue doing so in the coming years. I would also like to thank Professor Julian Shun who guided me on the GraphIt project. Julian's critical insights into graph applications and valuable feedback accelerated the project and helped a lot with this thesis.

I would also like to thank Yunming Zhang, a mentor and great friend who has guided me with the project and my research in general. Yunming's expertise in the field and laser focus really inspires me to be a better researcher. He has spent countless hours with me debugging code, writing papers and providing constant feedback and the project wouldn't have been complete without his help. Even outside work, he has always been there to listen and support me.

I would like to thank Changwan Hong and Shoaib Kamil who have provided valuable feedback on the GraphIt project and paper writing. I have learnt a great deal about optimizing and debugging GPU code from Changwan. I would also like to thank all the other collaborators on the project, Tugsbayasgalan Manlaibaatar and Claire Hsu who have been great to work with. I would also like to thank all the COMMIT group members for the amazing discussions and making my graduate school more enjoyable.

I want to express my gratitude to my parents, Sangita Brahmakshatriya and Rajendra Brahmakshatriya and my sister Jai for raising me to be the person I am today. Their sacrifice, love and support have a major contribution in my success. Thank you for always believing in me. Lastly, I would want to thank my close friends, Pratik Bhatu and Saurabhchand Bhati who have been beside me through thick and thin.

THIS PAGE IS INTENTIONALLY LEFT BLANK

Contents

THIS PAGE IS INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE IS INTENTIONALLY LEFT BLANK

List of Tables

Chapter 1

Introduction

Graph processing is at the heart of many modern applications, such as recommendation engines [18, 70], social networks [58, 11], and map services [53]. Achieving high performance is important because these applications often need to process large graphs with trillions of edges [40] or have strict latency requirements [18].

However, the performance of graph programs is notoriously difficult to optimize [76]. Graph programs exhibit irregular memory access patterns that are difficult to execute efficiently on modern hardware platforms, which are optimized for regular memory accesses. Performance bottlenecks of graph programs depend on the algorithm, the size and structure of the input graphs, and the underlying hardware.

1.1 Motivation

In the past, many diverse hardware platforms have been used to implement and execute graph applications. But no single hardware platform performs best for all graph applications. Some applications perform better on CPUs and others perform better on GPUs or Domain-Specific Accelerators (DSAs). Shared-memory CPUs have out-of-order execution, which helps hide the long latency of irregular memory accesses that miss in the last-level cache. CPUs also have larger memories than GPUs and other accelerators, which enables processing larger graphs. By contrast, GPUs have up to an order of magnitude more compute power and memory bandwidth than CPUs [51] and can better exploit the data parallelism of some graph programs when

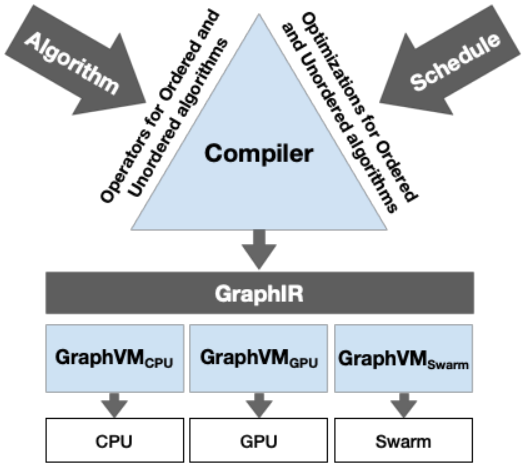

Figure 1-1: The Unified Graph Framework (UGF) provides a three-way decoupling between algorithm, schedule, and target hardware.

the graph fits in the GPU memory.

A large number of Domain-Specific Accelerators for sparse computations have emerged recently [22, 30, 57, 37, 14, 46, 2, 69, 3]. They provide hardware features such as efficient speculative execution that can drastically improve graph performance. However, the program must be transformed specifically for each DSA to take advantage of the new hardware features, yet it is infeasible to build a new compiler for each DSA. As a result, building a portable compiler infrastructure is crucial to exploit the diverse hardware of different DSAs.

Existing graph processing frameworks cannot achieve high performance across multiple hardware platforms with one unified programming abstraction. To get the highest performance, CPU and GPU processing libraries have adopted abstractions and optimizations specific to their hardware platforms [9, 42, 66, 60, 68]. Graph domain-specific languages (DSLs) [27, 20, 1], such as GraphIt [76, 74], are either unable to support platforms other than CPUs or slower than state-of-the-art libraries on the other platforms. Existing work cannot easily incorporate hardware-specific optimizations while maintaining a unified programming model.

1.2 Contributions

To achieve portability across CPUs, GPUs, and DSAs, we need to separate hardware-independent and hardware-specific optimizations. In this thesis, we propose a new

domain-specific intermediate representation (IR), Graph Intermediate Representation (GraphIR), to encode hardware-independent optimizations and serve as a high-level interface to different hardware backends. This way, hardware-independent optimizations can be reused across backends, and high-level data structures and operators can be mapped to different efficient hardware-optimized implementations. For example, the DeltaStepping algorithm for solving the single-source shortest paths problem needs priority queues to track the order of active vertices [44]. On CPUs, high-performance priority queue implementations use thread-local buckets [74, 8]. On GPUs, it is more efficient to compute the next bucket in the priority queue on the fly [42, 68]. And on Swarm hardware [30], priorities are maintained implicitly by using speculative tasks ordered with timestamps.

We also present a novel graph processing framework, the Unified Graph Framework (UGF), that unifies the creation of compiler backends for different architectures. The framework enables users to write high-performance graph algorithms once and run across GPUs, CPUs, and DSAs. UGF is built on top of the GraphIt DSL [76, 74], which decouples the algorithm from the performance optimizations (schedules) for graph algorithms. UGF utilizes a new scheduling language that combines load balancing, edge traversal direction, active vertex set creation, active vertex set ordering, and kernel fusion optimizations on GPUs and can be extended to support CPU and DSA-specific optimizations.

As shown in Figure 1-1, the UGF compiler performs various analyses and lowering passes to generate GraphIR, and the GraphIR is lowered into code for different architectures using an architecture-specific Graph Virtual Machine (GraphVM). GraphVMs perform hardware-specific transformations and code generation.

1.3 Thesis organization

In Chapter 2, we will talk about the GraphIt DSL compiler on which UGF is built. We will also talk about the optimization space for GPUs and the Swarm architecture. This will setup the stage for the new scheduling language that we have built.

In Chapter 3, we will delve deeper into the details of the extensible scheduling

language and how it is implemented for the GPU and Swarm architecture.

Chapter 4 talks about the details of the design of the GraphIR and GraphVMs. We will also present details about the arguments and the common metadata associated with the key GraphIR nodes (EdgeSetIterator and VertexSetIterator) and how this metadata can be used for optimizations.

Chapter 5 gets into the details of how the GraphIt DSL compiler is extended to separate all the transformations and analysis before the GraphIR generation. We will also present the details and nuances in the implementation of the GraphVMs for GPUs and Swarm. In this chapter we will also present two new performance optimizations that we have implemented: EdgeBlocking and ETWC. Finally we will also talk about how an autotuner can be built to automatically explore the vast scheduling space exposed to the programmer.

In Chapter 6, we evaluate our implementations against the other state of the art frameworks. We will also evaluate the effectiveness of our new optimizations by comparing the performance with them enabled and disabled.

Chapter 7 summarizes all the work and talks about some future directions for research.

Chapter 2

Background

The UGF framework is built as an extension to the GraphIt DSL compiler. UGF reuses the algorithm specification, parser, internal classes and some graph domain optimizations from GraphIt. In this chapter we will first provide some background about the GraphIt DSL compiler and it's algorithm language which is used to specify a program that operates on a graph, without needing to supply low-level details about how to execute it. We then contextualize this work by describing the tradeoffs inherent in executing graph algorithms on different hardware platforms, and the large space of existing hardware-independent and hardware-specific optimizations.

2.1 GraphIt DSL compiler

The main contribution of this thesis, UGF is built as an extension to the GraphIt high-performance DSL compiler for graph analytics reusing its algorithm specification, parser, CPU scheduling language and code generator and a lot of the existing transformation and analysis passes.

GraphIt achieves consistent high-performance across different algorithms and graphs while offering an easy-to-use high-level programming model for CPUs. GraphIt achieves this by decoupling the algorithm specification from optimization strategies for graph applications.

Because many graph applications require different optimization techniques, users normally have to try out a large set of such techniques to determine the best implemen-

tation for a particular algorithm and graph input. Implementing all these variations is not only time-consuming, but also requires whole program rewrite and careful debugging to ensure correctness. GraphIt solves this problem by separating the high-level algorithms from optimization decisions. Users specify graph algorithms using the algorithm language involving just high-level operations on sets of vertices and edges. They then use a separate scheduling language to compose different optimizations. This scheduling language supports a large space of optimization techniques, such as edge traversal direction, data layout, parallelization, cache efficiency and NUMA which are essential for high-performance on CPUs.

The GraphIt compiler has three main components: a frontend that scans and parses the high-level algorithm and the specific optimization chosen by the user, a midend that analyses and transforms the program with inputs from the scheduling decisions via multiple lowering passes to ensure correctness, and a backend that generates high-performance C++ implementations to run on multi-core CPUs.

But since the GraphIt DSL compiler is built only to target CPUs, there is no separation of transformations which are specific to CPUs and ones which can be applied to different architectures. This makes adding a new backend to support targets like GPUs or DSAs very tricky.

2.2 Algorithm Language

In this section we will describe GraphIt's algorithm specification language which UGF inherits. This algorithm language operates on vectors, vertex and edge sets, using functional operators over the sets and priority queue abstractions. The key purpose of the algorithm language is to allow programmers to specify the graph program without needing to specify performance-related low-level details, such as whether to use atomic synchronizations or which data structures to use for abstractions such as priority queues. Because UGF inherits the exact same algorithm language, programmers can specify a program using the algorithm language once, and specify details for performance on different platforms separately using the scheduling language.

Figure 2-1 shows an example of breadth-first search (BFS) using the algorithm

```
1 element Vertex end
2 element Edge end
3 const edges : edgeset{Edge}(Vertex,Vertex) = load(argv[1]);
4 const vertices : vertexset{Vertex} = edges.getVertices();
5 const parent : vector{Vertex}(int) = -1;
6
7 func toFilter(v : Vertex) -> output : bool
8   output = (parent[v] == -1);
9 end
10
11 func updateEdge(src : Vertex, dst : Vertex)
12   parent[dst] = src;
13 end
14
15 func main()
16   var frontier : vertexset{Vertex} = new vertexset{Vertex}(0);
17   var start_vertex : int = atoi(argv[2]);
18   frontier.addVertex(start_vertex);
19   parent[start_vertex] = start_vertex;
20   #s0# while (frontier.getVertexSetSize() != 0)
21     #s1# var output : vertexset{Vertex} =
22       edges.from(frontier).to(toFilter).
23         applyModified(updateEdge, parent, true);
24     delete frontier;
25     frontier = output;
26   end
27   delete frontier;
28 end
```

Figure 2-1: BFS program written in the UGF algorithm language.

language. BFS constructs a *BFS tree* from a single source vertex, where each reachable vertex from the source has a parent in the previous level. The user first initializes the element types, vertexset, edgeset, and parent vector in Lines 1–5, then specifies the `updateEdge` function (Line 12), which updates parents of newly-visited neighbor vertices. The `toFilter` function is a boolean function that filters out vertices that are already visited by checking if the parent value of the vertex is set to -1, which is the initial value (Line 8). In the `main` function, the program first initializes a new vertexset, `frontier`, (Line 16) and then adds a `start_vertex` to the frontier. On each round of the while loop, the algorithm applies the `updateEdge` function to edges whose destination vertices have not yet been visited (that is, the ones where the `toFilter` function (Line 23) returns true). The `applyModified` operator tracks which destination vertices' `parent` fields have been updated and adds them to the `output` vertexset. The program then deletes `frontier`, and then sets `frontier` to `output` (Lines 24–25). The while loop iterates until the `frontier` becomes empty (Line 20); once the while loop terminates, the parents of all reachable vertices are set, thus inducing the BFS tree.

2.3 Hardware Tradeoffs

One motivating reason for separating the algorithm language from low-level optimizations in UGF is that different optimizations are suitable for different algorithms, graph input and target hardwares. On one hand, CPUs have larger memories and out-of-order execution, which can hide the long latencies of irregular accesses to a large main memory and enable processing larger graphs. On the other hand, GPUs have up to an order of magnitude more computing power and memory bandwidth than CPUs [51] and can better exploit data parallelism in graph programs. We show in Chapter 6 that latency-sensitive algorithms with limited parallelism, such as SSSP on road graphs, run 2× faster on CPUs than on GPUs, whereas iterative graph processing algorithms that operate on a large number of vertices or edges in parallel, such as PageRank, perform up to 10× better on GPUs as long as the graph fits in GPU global memory.

Recent work has also proposed graph accelerators, such as the Swarm architecture [30], which adds hardware support for fine-grained task parallelism to a CPU. Swarm can achieve order-of-magnitude improvements in scalability over conventional CPUs on some graph algorithms by using dedicated hardware task queues and speculative execution to distribute tasks across hundreds of simple cores [29, 63, 32]. However, code must be optimized specifically to exploit the unique hardware acceleration features in such new architectures, in order to obtain any advantage from them.

To obtain the highest possible performance on a specific graph algorithm for a specific platform, programmers must utilized a combination of hardware-independent and hardware-specific optimizations.

2.4 Hardware-Independent Optimizations

Naive implementations of graph algorithms are orders of magnitude slower than optimized implementations [76, 43, 68, 42]. These optimizations are not always beneficial for different types of algorithms and graphs. As a result, the programmers often have to try multiple combinations of these optimizations to produce the best

implementation.

Some of the optimizations are hardware-independent as they either operate on the algorithm level or exploit features that are common across different hardware. UGF makes analyses and lowering passes for the hardware-independent optimizations reusable across different hardware backends. Here we give some examples of hardware-independent optimizations. We measure work-efficiency by the total number of instructions executed.

2.4.1 Direction-Optimization

When traversing edges in a graph, we can update vertex data in either push or pull mode. In the push mode, each source vertex updates their neighboring destination vertices in parallel. In the pull mode, each vertex reads incoming neighbors' data and updates itself in parallel. Compared to the pull mode, the push mode is more work-efficient, but incurs atomic synchronization overheads and has less parallelism. The direction-optimization switches between the push and pull modes depending on the size of the frontier [6, 60, 10].

2.4.2 Active Vertexset Data Layout

Vertexsets can have different data layout choices. A subset of vertices can be stored as a boolean array or bitvector array, where each vertex present has a true value. A vertexset can also be stored as a sparse array of IDs of vertices present in the set. Different layouts obviously have different performance charateristics and are suitable for different graph inputs and algorithms.

2.4.3 Active Vertex De-duplication

Since active vertices may share common neighbors, some vertices can be added multiple times to the next frontier, which can cause redundant computation and even lead to incorrect results. To perform deduplication efficiently, we use different implementation strategies, such as a bitvector or byte array, to keep track of vertices already in the output frontier.

2.4.4 Active Vertexset Processing Ordering

The active vertices can be processed according to a priority-based ordering, which can significantly improve the work-efficiency for ordered graph algorithms such as single-source shortest paths, but at the expense of lower parallelism [44, 74, 42]. There is a plethora of variations in implementations of the priority queues that offer tradeoff between work-efficiency, parallelization and synchronization.

2.4.5 Parallelization

We can exploit vertex-level or edge-level parallelism when performing data-parallel operations on edges. The vertex-level approach generates less parallelism than the edge-level approach, but incurs less synchronization overhead. The compiler uses this information to insert appropriate synchronization primitives. Different hardware have specific ways to implement vertex-level or edge-level parallelism.

2.5 GPU-Specific Optimizations

In addition to the hardware-independent optimizations, UGF also takes advantage of the features of the underlying hardware to get the highest level of performance. In this section, we describe the GPU-specific optimizations that we incorporated in UGF. These optimizations are implemented as both compiler transformations and runtime libraries as we will describe in Chapter 3 and Chapter 5.

2.5.1 Active Vertexset Creation

There are different ways to create the output frontier, including fusing it with the edge traversal or having a separate operator [42]. This can be combined with different representations of the vertexset, such as a sparse array (only storing indices of active vertices), a bitmap (one bit per vertex for all vertices), or a byte map (one byte per vertex for all vertices). The fused mode sacrifices parallelism but improves work-efficiency. The bitmap array has better locality but creating it requires atomic synchronization overhead.

2.5.2 Kernel Fusion across Iterations

The iterative nature of graph algorithms can be expressed with a loop. A GPU kernel can be launched in each iteration of the loop. However, if there is not enough work per iteration, the kernel launch overhead can dominate the running time. To alleviate this overhead, the loop can be moved into the kernel so that the GPU kernel is launched only once [52]. This improves work-efficiency but potentially sacrifices parallelism due to worse load-balancing from a single kernel launch.

2.5.3 Load-Balancing

Different load-balancing schemes make trade-offs between parallelism and work-efficiency to various degrees. Warp Mapping (WM) and CTA Mapping (CM) [42, 33] divide the active vertices evenly across different warps and CTAs. Each thread processes an equal number of vertices. WM and CM achieve high parallelism but sacrifice work-efficiency due to overheads in partitioning the active vertices. Assigning each thread the same number of edges (STRICT) incurs an even higher overhead than WM and CM, but ensures that each thread in the grid processes the same number of edges. TWC splits active vertices into buckets processed by threads, warps, or CTAs based on their degrees [43, 68]. TWC has a small runtime overhead (high work-efficiency) but a lower degree of parallelism compared to WM, CM, and STRICT. Vertex-Parallel (VP) simply maps each vertex to a thread in the grid. VP has the lowest runtime overhead, but potentially the worst load balancing. In this thesis, we also introduce a new load balancing optimization, Edge-based Thread Warps CTAs (ETWC), that can achieve load balancing as good as TWC while reducing the runtime overhead compared to TWC. More details are provided in Section 5.2.

2.5.4 GPU Cache Optimization

Long-latency irregular memory access is a major performance bottleneck in many graph algorithms. We introduce a new optimization, EdgeBlocking, which exploits the fast shared GPU L2 cache by partitioning the graph (Section 5.2).

2.6 CPU-Specific Optimizations

UGF inherits NUMA and cache partitioning for graphs on CPUs to take advantage of local DRAM and last level cache (LLC) in the systems outlined in existing work [76, 75]. These optimizations significantly improve the locality of memory access. For priority-based algorithms, we utilized CPU-specific optimizations on improving the work-efficiency and reducing the synchronization overhead to maintain priority-based ordering introduced in previous work [74]. These optimizations make use of large amounts of memory available to each core in CPUs to maintain thread-local priority-based data structures.

2.7 Swarm-Specific Optimizations

Swarm provides hardware support for parallelization by relying on order as the main synchronization primitive. This hides the effects of concurrency fom software. Swarm programs consist of tasks that are ordered by timestamps. Tasks can read and write memory and spawn more tasks. Each task is given a timestamp when it is spawned. Swarm guarantees that tasks appear to run atomically and in timestamp order. To scale, Swarm hardware executes tasks in parallel and out of order. To preserve ordered semantics, tasks execute speculatively, and are aborted and re-executed if their memory accesses produce an order violation.

This execution model is a natural fit for priority-based algorithms, where a task's priority can serve as its timestamp. Swarm's speculative execution uncovers more parallelism than conventional CPUs and GPUs by executing tasks with different priority levels in parallel. However, this can incur performance penalties because of aborts. To reduce the cost of aborting large tasks, they can be broken down into small tasks [71]. Swarm's hardware support makes task spawns as cheap as a few clock cycles, enabling small tasks. This creates new opportunities and challenges for a compiler to optimize performance by balancing parallelism and work-efficiency.

Chapter 3

Scheduling Language

We propose an extensible scheduling language design for UGF. We first describe the object-oriented design for the scheduling language, then describe in more detail the novel GPU scheduling language that allows users to combine the load-balancing, traversal direction, active vertexset management, and work-efficiency optimizations described in Chapter 2. A similar discussion is also presented regarding the Swarm scheduling language. The extensible scheduling language design makes it easy to implement a new scheduling language for other backends in future.

Scheduling Language Interface. We designed a common scheduling language interface for the compiler to ensure that the scheduling language for a new hardware can be added without modifying the hardware-independent analysis and lowering passes in the compiler. A scheduling language constructs a scheduling object that keeps track of the various performance optimizations applied. The scheduling objects for different backends extend the same base scheduling object class that the compiler uses to acquire information for hardware-independent optimizations outlined in Section 2.4. For example, the compiler will use the scheduling object to determine the direction of traversal and whether it exploits vertex-level or edge-level parallelism. Each scheduling language can also include hardware-specific optimizations, which are passed directly to the hardware backend.

3.1 GPU Scheduling Language

We introduce a novel scheduling language for GPU graph analytics that allows programmers to easily combine together different load balancing, edge traversal direction, frontier creation, kernel fusion, and other optimizations. Users can search through a large space of GPU optimizations without changing the algorithm specification. The scheduling language uses two main types of scheduling objects: `SimpleGPUSchedule` and `HybridGPUSchedule` to specify the optimizations.

The `SimpleGPUSchedule` object directly controls the scheduling choices related to load-balancing, traversal direction, active vertexset management, and work-efficiency. As shown in Table 3.1, the objects have six `config*` functions. The programmer can use these functions to specify the load-balancing strategy, the edge traversal direction, the output frontier creation strategy, whether or not de-duplication is enabled, the delta value for priority queues, and whether or not Kernel Fusion is applied to particular loops. Some of these functions also have optional parameters that the programmer or an autotuner can use to further tune the schedule.

The `HybridGPUSchedule` objects combine two `SimpleGPUSchedule` objects with some runtime condition. The two `SimpleGPUSchedule` objects can be entirely different, using different load-balancing schemes, frontier creation types, traversal directions, etc. Depending on whether the runtime condition evaluates to true or false, one of the two `SimpleGPUSchedule` objects is invoked. Currently, the `HybridGPUSchedule` is only supported with the `edges.apply` operator. This API enables the programmer to implement complex optimizations like direction-optimization by combining two `SimpleGPUSchedule` objects. The details of the `HybridGPUSchedules` are provided in Table 3.2.

The programmer uses the `applyGPUSchedule` function of the `program` object to apply either a `SimpleGPUSSchedule` or a `Hybrid GPUSchedule` object to a statement identified by a label. The programmer supplies the label of the statement as a string.

Scheduling for BFS. Figure 3-1 shows how to create and apply a direction-optimizing schedule for the BFS program, which switches between a push-based and a pull-based execution strategy based on the size of the frontier, as discussed in Section 2.4. On

Member function	Parameters	Description
`SimpleGPUSchedule`	*`SimpleGPUSchedule s0`*	Creates a new SimpleGPUSchedule object with an optional object to copy all the options from
Scheduling functions for `edges.appply`		
`configLoadBalance`	`load_balance_type`, *`blocking_type`*, `int32_t blocking_size`	Configure the load balance scheme to be used from one of `VERTEX_BASED`, `CM`, `WM`, `STRICT`, `EDGE_ONLY`, `ETWC` and `TWC`. Optionally enable blocking and set blocking size
`configDirection`	`direction`, *`frontier_rep`*	Configure the direction of updates of vertex data between `PUSH` and `PULL`. Optionally also configure the representation of the frontier (`BOOLMAP` or `BITMAP`) when `PULL` direction is used.
`configFrontierCreation`	`frontier_type`	Configure how the output frontier will be created. Options are `FUSED`, `UNFUSED_BOOLMAP` and `UNFUSED_BITMAP`
`configDeduplication`	`dedup_enable`, `dedup_type`	Configure if deduplication needs to be enabled when producing the output frontier (`ENABELD`). Strategy can be `MONOTONIC_COUNTERS`, `BITMAP`, `BOOLMAP`
Scheduling functions for `edges.appplyUpdatePriority`		
`configDelta`	`int32_t delta`	Configure the delta value to be used when creating the buckets in the priority queue
Scheduling functions for `while(condition)`		
`configKernelFusion`	`enable_fusion`	Configure if kernel fusion is `ENABLED` or `DISABLED` for this while loop

Table 3.1: Description of the SimpleGPUScheduling type and associated config functions. Optional parameters are shown in *italics*.

Function	Parameter	Desription
`HybridGPUSchedule`	`hybrid_criteria`, `float threshold`, `SimpleGPUSchedule s1`, `SimpleGPUSchedule s2`	Create a HybridGPUSchedule object by combining two SimpleGPUSchedule objects with a runtime condition (currently can be only `INPUT_VERTEXSET_SIZE`) and a threshold

Table 3.2: Description of the HybridGPUScheduling type and associated config functions.

Lines 2–6, we first declare a `SimpleGPUSchedule` object `s1` for the push direction. We configure the load-balancing strategy to be ETWC, direction to be `PUSH`, and select the frontier creation to be `UNFUSED_BITMAP`. Then we declare another `SimpleGPUSchedule s2` for the pull direction on Line 7 that first makes a copy of `s1`. For the pull-based schedule object, the programmer configures `VERTEX_BASED` load-balancing in the `PULL` direction. On Line 11, we declare a `HybridGPUSchedule`, which selects between `s1` and `s2` based on the size of the input frontier specified with `"argv[3]"`. Finally, we apply this schedule to the `applyModified` operator (Line 23 of Figure 2-1) in the algorithm language using the `applyGPUSchedule`. `"s0:s1"` is a scoped label that references the `applyModified` operator (with the label `"s1"`) within the while loop (with the label `"s0"`).

```
schedule:
    SimpleGPUSchedule s1;
    s1.configDeduplication(ENABLED);
    s1.configLoadBalance(ETWC);
    s1.configDirection(PUSH);
    s1.configFrontierCreation(UNFUSED_BITMAP);
    SimpleGPUSchedule s2 = s1;
    s2.configLoadBalance(VERTEX_BASED);
    s2.configDirection(PULL, BITMAP);
    s2.configDeduplication(DISABLED);
    HybridGPUSchedule h1 (INPUT_VERTEXSET_SIZE,
      "argv[3]", s1, s2);
    program->applyGPUSchedule("s0:s1", h1);
```

Figure 3-1: Example of a schedule that can be applied to the BFS program. This schedule is suitable when running BFS on graphs on a GPU with power-law degree distributions, such as social graphs.

```
schedule:
    SimpleGPUSchedule s1;
    s1.configDeduplication(DISABLED);
    s1.configLoadBalance(ETWC);
    s1.configDirection(PUSH);
    s1.configFrontierCreation(FUSED);
    program->applyGPUSchedule("s0:s1", s1);
    SimpleGPUSchedule s0;
    s0.configKernelFusion(ENABLED);
    program->applyGPUSchedule("s0", s0);
```

Figure 3-2: Example of a schedule that can be applied to the BFS program. This schedule is suitable when running BFS on high-diameter graphs with low degree, such as road graphs on a GPU.

3.2 Scheduling on CPUs and Swarm

Due to space constraints, we summarize the scheduling extensions for CPUs and Swarm in a single section. We refactor the CPU scheduling language in GraphIt [76, 74] to implemente the abstract sheduling classes mentioned above. This way the language can share the ability to cofigure the target-independent analyses and transformations in UGF like choosing the edge traversal direction, whether de-duplication is enabled, and whether the iteration is vertex-parallel or edge-parallel. The CPU-specific optimizations described in Section 2.6 are passed directly to the CPU backend (CPU GraphVM).

Similar to the CPU and the GPU GraphVMs, the Swarm GraphVM has its own scheduling options apart from the high-level schedules. Since the Swarm architecture speculatively executes tasks with timestamps, the GraphVM needs to divide the program into tasks. The Swarm GraphVM lets the user specify scheduling commands to split the writes to vertex data into separate tasks that hardware can cheaply abort and re-execute. This scheduling option provides a tradeoff between the amount of work wasted because of aborts and the overhead of launching more fine-grained tasks. The Swarm GraphVM also lets the user set the number of cores used to execute the program, exposing a tradeoff between parallelism and the amount of aborts.

Chapter 4

Graph Intermediate Representation (GraphIR)

In this chapter, we present GraphIR, an intermediate representation that decouples algorithm specification and hardware-independent optimizations from hardware-specific optimizations. GraphIR enables us to build reusable intermediate representations, program analyses, and lowering passes shared across different hardware platforms. This greatly reduces the amount of effort needed to support a new backend (GraphVM) for GraphIt.

4.1 GraphIR Representation

The GraphIR is composed of variables, functions, and instructions. Each variable, function, or instruction carries both arguments and metadata as shown in Table 4.1. The arguments capture all of the information derived from the algorithm specification. The metadata capture information related to the performance optimizations. The GraphIR metadata includes hardware-independent optimizations discussed in Section 2.4, such as the direction of traversal and whether the operator exploits edge-level parallelism.

To perform hardware-specific transformations and code generation, each backend implements an abstract machine (GraphVM) to optimize and run the GraphIR, similar to the Java VM or LLVM. We provide more details about the GraphVMs in Chapter 5.

GraphIR Types

Type	Description
`EdgeSet`	Graph data type. Can be weighted or unweighted. Can have `COO` or `CSR` representation.
`VertexSet`	Type to hold a set of vertices. Can have `SPARSE`, `BITMAP` or `BOOLMAP` representation.
`Function`	Top level function definition type. Functions can be annotated as `DEVICE`, `HOST` or both.

GraphIR Instructions

Instruction	Arguments	Metadata
`VertexSetIterator`	`VertexSet input_vset`, `Function apply_f`	`bool is_all_vertices`: Specifies if iterates over all the vertices. `bool is_parallel` Specifies whether it is a parallel iterator
`EdgeSetIterator`	`EdgeSet input_graph`, `VertexSet input_vset`, `VertexSet output_vset`, `Function apply_f`	`bool is_all_edges`: Specifies if it iterates over all the edges. `bool requires_output`: Specifies if it generates an output set. `bool apply_deduplication`: Specifies if the generated output vertex requires an explicit deduplication step. `VertexSetRepresentation output_representation`: Specifies the representation of the output frontier. `bool can_reuse_frontier`: Specifies if the output frontier can reuse the input frontier data structure. `bool is_edge_parallel`: Specifies if it process edges in parallel. `DirectionType direction`: Specifies `PUSH` or `PULL` direction. `VertexSetRepresentation pull_input_frontier`: Specifies the representation for the input frontier in the `PULL` direction.

Table 4.1: The key instructions in the GraphIR, the EdgeSetIterator and VertexSetIterator

For each GraphVM, the GraphIR metadata can be extended easily to support the hardware-specific optimizations discussed in Section 2.5 and Section 2.6. Hardware GraphVM developers can add custom fields to the flexible metadata to perform hardware-specific analyses, optimizations, and code generation that require some capability information specific to the hardware. To ensure correctness, the compiler needs to follow the specifications of arguments and hardware-independent metadata.

The two most important instructions in the GraphIR are the `EdgeSetIterator` and `VertexSetIterator` instructions, which are shown in Table 4.1. The `EdgeSetIterator` iterates through all or a subset of the edges of a graph and invokes a function on each edge. The arguments of `EdgeSetIterator` specify the graph (`input_graph`), input frontier vertexset

(`input_vset`), output frontier vertexset (`output_vset`), and the user-defined function that works on the edges (`apply_f`). These arguments are derived from the operators in the algorithm specification. The instruction also has metadata for generating optimized implementations, such as choosing the input/output frontier representations, edge traversal direction, deduplicating the output frontiers, or generating specialized code if the edge set representation is dense. The `VertexSetIterator` iterates over all of the vertices in a frontier, and similarly has arguments and metadata for optimizations.

Apart from these key instructions, the GraphIR has instructions for data structure allocation both on the host and on the device, general arithmetic and reductions, and program control flow. The GraphIR abstraction is at a higher level than nested for-loops because not all hardware can work with the for-loop abstractions. For example, GPUs can spawn different threads for the outer loops or a set of threads for both the loops combined.

Architectures with these features make use of the metadata attached to the instructions to implement various optimizations. For example, GPUs, which have a hierarchy of threads, can implement different load-balancing strategies to efficiently process vertices with varying degrees. CPUs and GPUs both have multiple levels of memory, which enables cache-blocking optimizations, such as EdgeBlocking.

4.2 GraphIR for BFS

We use snippets of code for BFS to illustrate the GraphIR representation as shown in Figure 4-1. The GraphIR for BFS has the `updateEdge` function that is applied to each edge in the `EdgeSetIterator`. Analyses, such as dependence analysis for inserting atomics and liveness analysis for frontier reuse, are hardware-independent and are reusable across the different hardware backends. Line 4 shows that the high-level compiler inserts an `AtomicCompareAndSwap` after the dependence analysis. The `AtomicCompareAndSwap` instruction can be mapped to different underlying implementations for different backends (GraphVM). For CPUs, this can be mapped to the built-in compare-and-swap (CAS) instruction. For GPUs, the atomic operation can be implemented efficiently using warp shuffling instructions for easy synchronization and exchange of values among threads. The `main`

```
...
Function updateEdge (int32_t src, int32_t dst,
    VertexSet output_frontier, {
    bool enqueue = AtomicCompareAndSwap(parent[dst], -1, src),
    If (enqueue, {
        EnqueueVertex<format=SPARSE>(output_frontier, dst)
    }, {})
})
Function main (int32_t argc, char* argv[], {
    ...
    WhileLoopStmt<needs_fusion=true>(VertexSetSize(frontier), {
        EdgeSetIterator<requires_output=true,
            can_reuse_frontier=true,
            direction=PUSH,
            is_edge_parallel=true>(
            edges, frontier, output, updateEdge, toFilter),
        AssignStmt(frontier, output)
    }),
    ...
})
```

Figure 4-1: The GraphIR generated by the compiler for the BFS algorithm and the schedule in Figure 3-2. Parameters to instruction are specified in `()` and metadata is specified in `<>`. Some of the metadata is omitted for brevity.

function has an `EdgeSetIterator` instruction inside the `WhileLoopStmt`. This instruction has `can_reuse_frontier` set to `true` because the liveness analysis determines that the output frontier can reuse the memory allocated for the input frontier. This is a hardware-independent optimization attached as metadata.

The GraphIR is also extensible to support hardware-specific optimizations. For the GPU GraphVM, multiple kernel launches can be fused into a single kernel launch as explained in Section 2.5. The GPU GraphVM extends the metadata of `WhileLoopStmt` with a flag `needs_fusion` and sets it to `true` with a hardware-specific analysis compiler pass. This indicates that the schedule has prescribed the fusion of all of the operator calls inside the loop into a single kernel. The GraphVM will then generate a fused kernel if the hardware supports it.

Chapter 5

Compiler Implementation

This chapter describes two hardware-independent passes in UGF—liveness analysis for frontier reuse and dependence analysis for atomic instruction insertion—and provides more details on the three GraphVMs for GPU, Swarm, and CPU.

5.1 Hardware-independent passes

The hardware-independent analysis and transformation passes are implemented in the UGF compiler before generating the GraphIR and make use of the interface from the abstract scheduling classes to query the hardware-independent scheduling decisions.

5.1.1 Liveness Analysis for Frontier Reuse

In algorithms such as BFS and SSSP, the `edges.applyModified` (Figure 2-1 Line 23) operator allocates memory for the new output `VertexSet` that it produces. We can avoid the overhead of allocation and deletion if we reuse the memory allocated for `frontier` for `output`. To apply this optimization in a general way, we implement a liveness analysis pass in the high-level compiler that finds `applyModified` operators where the input `VertexSet` is deleted before it is used again and sets the `can_reuse_frontier` metadata in the generated GraphIR as seen in Figure 4-1 Line 13. The compiler also removes the corresponding `delete` operator.

```
void __device__ updateEdge(
    int src, int dst, VertexFrontier output_frontier) {
    bool enqueue = CAS(&parent[dst], -(1), src);
    if (enqueue) {
        enqueueVertexSparseQueue(
            output_frontier.d_sparse_queue_output,
            output_frontier.d_num_elems_output, dst);
    }
}
void __global__ fused_kernel_body_1(void) {
    ...
    while (device_builtin_getVertexSetSize(local_frontier)) {
        vertex_set_prepare_sparse_device(local_frontier);
        local_output = local_frontier;
        ETWC_load_balance_device<
            int, updateEdge, AccessorSparse, true_function> (
            edges, local_frontier, local_output);
        swap_queues_device(local_output);
        local_output.format_ready = VertexFrontier::SPARSE;
        local_frontier = local_output;
    }
    ...
}
int __host__ main(int argc, char* argv[]) {
    ...
    cudaLaunchCooperativeKernel(
        (void*)fused_kernel_body_1,
        NUM_CTA, CTA_SIZE, no_args);
    ...
}
```

Figure 5-1: CUDA code generated for the BFS GraphIR with the kernel fusion optimization.

5.1.2 Dependence Analysis for Inserting Atomic Instructions

Depending on the order of iteration over the edges, updates to shared memory may introduce data races because the algorithm language does not require the user to write atomic operations. The UGF compiler identifies which accesses inside the user-defined functions can have data races. For example, Line 12 of Figure 2-1 updates the `parent` field of the destination vertex, and if multiple threads update the same destination vertex, there is a data race. This decision depends not only on the algorithm representation but also on the schedules applied. For example, depending on whether the threads are iterating in the PUSH or the PULL direction, a race can appear at the access with `src` or `dst` vertex. The compiler performs a dependence analysis to identify shared accesses among different threads. These access are marked to have a SHARED access. After this, the compiler performs a read-write analysis to further identify access that updates memory using some condition or with a reduction. Finally, atomic accesses are inserted for those memory accesses that update memory and also have the SHARED flag set.

5.2 GPU GraphVM implementation

The GPU GraphVM uses the GraphIR as input to perform GPU-specific transformations and optimizations, such as load balancing and fusing kernel launches. The GPU GraphVM first performs a series of passes that analyze and optimize the GraphIR. It then generates CUDA C++ programs and executes them with the UGF GPU runtime library.

One of the first analyses that the GPU GraphVM performs is to determine which functions need to run on GPU and which need to run on the host CPU, since this affects the optimizations and code generation. Lines 1 and 10 of Figure 5-1 show functions compiled for the GPU and Line 24 shows code for the functions to be executed on the host. The GPU GraphVM also inserts appropriate instructions for data transfers between the GPU and the host.

The GPU GraphVM generates code in multiple passes. First, the kernels executed on the GPU are generated for each operator (`EdgeSetIterator` and `VertexSetIterator`). Any specialized kernels for fused loops are generated next. Finally, the host code and other device functions are generated.

5.2.1 Kernel Fusion Optimization

To apply kernel fusion, the GPU GraphVM first analyzes all loops to identify candidates that can be fused. The compiler then transforms statements in the body of the candidate loops. This optimization reduces the overhead incurred from launching a separate kernel for each `VertexSetIterator` or `EdgeSetIterator`, as explained in Section 2.5. As shown in Line 20 of Figure 2-1, we can add a label for the entire loop. The programmer can then enable Kernel Fusion as shown on Line 9 of Figure 3-2. With this schedule, the compiler must launch a single CUDA kernel for the entire loop. This means that all steps inside the body of the loop should be executed on the device. The GraphVM first performs an analysis to figure out if the body contains a statement that it cannot execute on the device. This decision is GPU-specific and hence can be done only in the GPU GraphVM. The GPU GraphVM then creates a single `__global__` kernel for

linesnumbered 1 Pseudocode for the implementation of ETWC load balancing strategy.

Input: Graph G in CSR format, Input Frontier (input_frontier), and three Queues Q[0...2], the number of threads to process edges of a vertex in three Stages Stage_gran[0...2]

```
1: idx = threadblockID * threadblockSIZE + threadID
2: Initialize Q[0 ... 2]
3: if idx < input_frontier.size then
4:     src_id ← getFrontierElement(input_frontier, idx);
5:     size ← G.rowptr[src_id + 1] − G.rowptr[src_id]
6:     start_pos ← G.rowptr[src_id]
7:     end_pos ← G.rowptr[src_id + 1]
8:     Stage_elt[2] ← ⌊size/Stage_gran[2]⌋ × Stage_gran[2]
9:     if Stage_elt[2] > 0 then
10:        Q[2].Enqueue({start_pos, start_pos + Stage_elt[2], src_id});
11:        start_pos ← start_pos + Stage_elt[2]
12:        size ← size − Stage_elt[2]
13:    Stage_elt[1] ← ⌊size/Stage_gran[1]⌋ × Stage_gran[1]
14:    if Stage_elt[1] > 0 then
15:        Q[1].Enqueue({start_pos, start_pos + Stage_elt[1], src_id});
16:        start_pos ← start_pos + Stage_elt[1]
17:        size ← size − Stage_elt[1]
18:    if size > 0 then
19:        Q[0].Enqueue({start_pos, end_pos, src_id});
20: sync_threads()
21: for i : 0, 1, 2 do
22:    while !isEmpty.Q[i] do
23:        if threadID%Stage_gran[i] == 0 then
24:            {start_pos, end_pos, src_id} ← Q[i].deQueue()
25:            Broadcast({start_pos, end_pos, src_id}, threadID ... threadID + Stage_gran[i] − 1)
26:        cooperative for eid in start_pos : end_pos − 1 do
27:            dst_id ← G.edges[eid]
28:            process_edge(src_id, dst_id)
```

the entire `WhileLoopStmt` with all the operations in the body inlined and inserts a single kernel launch inside the `main` function. The GraphVM also creates thread copies and `__global__` copies of local variables that are accessed inside the loops and makes sure that their accesses are coordinated. The GraphVM doesn't have to analyze the entire loop body because the high-level compiler has already annotated the variables used inside. Line 10 of Figure 5-1 shows one such `__global__` kernel for the `WhileLoopStmt` in the BFS application.

One of the challenges when generating code using kernel fusion is that different operators inside the body of the loop require different numbers of threads and CTAs, but the fused kernel has fixed number of threads and CTAs. To handle this, the code generator inserts a loop around the call to the operator so that a fixed number of CTAs simulate the behavior of more CTAs. All of the threads in the grid are synchronized after the execution of each operator.

5.2.2 Edge-based Thread Warps CTAs (ETWC)

ETWC further reduces the runtime overhead of TWC by performing TWC style assignment within each CTA instead of across all CTAs. Similar to CM, an equal number of vertices are assigned to each CTA, which can potentially hurt parallelism. Within each CTA, ETWC partitions edges of vertices into chunks that are processed by a thread, a warp, or the entire CTA. Each CTA processes chunks of different sizes in separate stages to increase the number of threads involved in the computations (i.e., a better parallelism) in each stage.

We show the pseudocode for the ETWC load-balancing strategy in Algorithm 1. Each thread has a unique ID (idx), and three queues ($Q[0]$, $Q[1]$, and $Q[2]$) are initialized on Line 3. In the first stage, $Stage\_gran[0]$ threads cooperatively process edges of a vertex. Similarly, in the second and third stages, $Stage\_gran[1]$ and $Stage\_gran[2]$ threads process edges of a vertex together. Note that the optimal values for $Stage\_size[0]$, $Stage\_size[1]$, and $Stage\_size[2]$ are dependent on the algorithm, graph, and platform. Lines 4–20 shows how to split the edges of a vertex into three chunks with division and modular operations to improve parallelism while the number of edges in the first stage is minimized. First, for a vertex (src_idx), we pick up as many as edges that is a multiple of $Stage\_size[2]$, which is stored in $Stage\_elt[2]$ on Line 9 and put them in $Q[2]$ (Line 11) if $Stage\_elt[2]$ is not 0 (Line 10). After that, $start\_pos$ and $size$ are adjusted. Similarly, the maximum number of edges, which is a multiple of $Stage\_size[1]$, is stored in $Q[1]$ (Lines 14–18). The remaining edges are moved to $Q[0]$ (Lines 19–20). The for-loop on Line 22 processes edges of vertices. For each stage, the representative thread ($threadId\%Stage\_size[i] == 0$) dequeues the 3-tuple of the corresponding queue and broadcasts it for $Stage\_size[i]$ threads to cooperatively process the corresponding edges in a cyclic fashion. We note that queues are managed in shared memory to reduce the overhead for memory operations (e.g., global memory atomic operations).

linesnumbered 2 Algorithm for preprocessing the input graph for applying the EdgeBlocking optimization. The algorithms takes as input a graph in the COO format and the blocking size. The output is a new graph in COO format with the edges blocked

Input: Number of vertices per segment N, Graph G in COO format
Output: Graph Gout in COO format

```
1: numberOfSegments ← ceil(G.num_vertices/N)
2: segmentSize[numberOfSegments + 1] ← 0
3: Gout ← new Graph
4: for e : G.edges do
5:     segment ← floor(e.dst/N)
6:     segmentSize[segment] ← segmentSize[segment] + 1
7: prefixSum ← prefix_sum(segmentSize)
8: for e : G.edges do
9:     segment ← floor(e.dst/N)
10:    index ← prefixSum[segment]
11:    Gout.edges[index] ← e
12:    prefixSum[segment] ← prefixSum[segment] + 1
13: Gout.numberOfSegments ← numberOfSegments
14: Gout.segmentStart ← prefixSum
15: Gout.N ← N
```

5.2.3 EdgeBlocking

We propose a new EdgeBlocking optimization that tiles the edges into a series of subgraphs to improve the locality of memory accesses. Algorithm 2 shows the steps for preprocessing an input graph to apply the EdgeBlocking optimization. The preprocessing is a two step process. The first for-loop (Line 4 of Algorithm 2) iterates through all of the edges and counts the number of edges in each subgraph. The algorithm then uses a prefix sum of these counts to identify the starting point for each subgraph in the output graph edges buffer. The second for-loop (Line 8 of Algorithm 2) then iterates over each edge again and writes it to the appropriate subgraph while incrementing that subgraphs counter. We apply the function `process_edge` to each edge as shown in Algorithm 3. The arguments to this function are the source vertex and the destination vertex. The pseudocode shown in Algorithm 3 is executed by each thread in a thread block. We use an outer for-loop (Line 1 of Algorithm 3) that iterates over each subgraph. Within each subgraph, the edges are then processed by all of the threads using a `cooperative for` (Algorithm 3 Line 4). All of the threads are synchronized between iterations over separate subgraphs to avoid cache interference. EdgeBlocking improves the performance of some algorithms by up to 2.94x as show in Table 6.7.

linesnumbered 3 Pseudocode for the implementation of the *edgeset.apply* operator with the EdgeBlocking optimization. Here the input graph is pre-processed and the edges are blocked. The function *process_edge* is applied to each edge in the graph. Notice that the EdgeBlocking optimization can be applied only when all the edges in the graph are being processed

Input: Graph G in COO format, Vertex data array V of type T

1: **for** $segIdx : G.numberOfSegments$ **do**
2: $start\_vertex \leftarrow G.N * segIdx$
3: $end\_vertex \leftarrow G.N * (segIdx + 1)$
4: **cooperative for** eid *in* $G.segmentStart[segIdx - 1] : G.segmentStart[segIdx]$ **do**
5: $e \leftarrow G.edges[eid]$
6: $process\_edge(e.src, e.dst)$
7: $sync\_threads()$

```
1 void updateEdge(int src, int dst) {
2     float &target = new_rank[dst];
3     float val = contrib[src];
4     swarm_spawn {
5         target += val;
6     }
7 }
8 ...
9 for (int _iter = 0; _iter < edges.num_edges; _iter++) {
10    int _src = edges.edge_src[_iter];
11    int _dst = edges.edge_dst[_iter];
12    updateEdge(_src, _dst);
13 };
```

Figure 5-2: Swarm C++ code generated from the Swarm GraphVM for the PageRank algorithm. A separate task is launched for each edge to be processed. Some of the code has been omitted for brevity.

5.3 Swarm GraphVM

Similar to the GPU GraphVM, we have also built a Swarm GraphVM to optimize the GraphIR for the Swarm architecture. This further demonstrates UGF's ability to target different kind of architectures. As explained in Section 2.7, the Swarm architecture uses task-based speculative execution to extract maximum parallelism from applications. We will now discuss how the GraphVM generates code for different types of applications. The Swarm GraphVM generates sequential C++ code (as shown in Figure 5-2), which the T4 compiler [71] converts into tasks for the Swarm hardware. T4 assigns tasks increasing timestamps to preserve sequential program order while extracting speculative parallelism. The T4 compiler automatically parallelizes loop nests by spawning a separate task for each iteration of each loop. The T4 compiler also takes annotations to further split the code into smaller tasks. One such annotation is shown on Line 4 in Figure 5-2.

For algorithms that iterate over all of the edges in the graph, a separate task is

created for each edge. As shown in the PageRank example on Line 9 of Figure 5-2, the GraphVM generates a loop that iterates over all of the edges. The T4 compiler then turns each loop iteration into a separate Swarm task. To achieve the maximum performance, the GraphVM further breaks each iteration install finer tasks so that speculative writes are performed in their own task, separate from non-speculative reads at the start of each loop iteration. This reduces the overhead when some writes are aborted and re-executed, since only the writes are re-executed again. The GraphVM does this by inserting a task boundary annotation as shown on Line 4 of Figure 5-2. The scheduling language for the Swarm GraphVM lets the user control these task boundaries. T4 gives unique timestamps to all tasks, thus recording the sequential code's program order. Thus, the rounds of PageRank have increasing timestamps, guaranteeing behavior equivalent to bulk-synchronous execution.

For algorithms that iterate over a dynamically determined frontier, we cannot launch all of the tasks upfront. Tasks are launched speculatively whenever a vertex is added to the frontier in the `EdgeSetApply` operator. SSSP is an example of one such program, where a separate task is created for each vertex, with timestamp equal to the priority from the priority queue. When a vertex's data is updated, a task is launched for each outgoing edge.

5.4 CPU GraphVM implementation

The CPU GraphVM generates high-performance C++ code to run on multi-socket, multi-core CPUs. We refactored the CPU-specific passes, code generator, and runtime libraries from existing work [76, 74] to be compatible with the new common scheduling interface described in Chapter 3. Apart from the CPU-specific transformations, the CPU GraphVM also generates data structures that are specific to the CPU backend, such as intermediate buffers for prefix sums or thread-local buckets for priority queues. Thread-local queues are only used by the CPU and not by the GPU because the GPU has an order of magnitude larger number of threads and lesser amount of total memory. An example of an optimization performed specifically by the CPU GraphVM is the bucket fusion optimization introduced in previous work [74]. This optimization reduces

the amount of synchronization required among threads after processing the buckets from the priority queue, and improves overall performance by up to 3×.

5.5 Autotuning

The UGF compiler exposes a large optimization space, with about 10^6 combinations of different schedules. Even without the hybrid schedules that involve two traversal directions, the compiler can generate up to 288 combinations of schedules for each direction (Table 6.10). On top of that, integer and floating-point parameters like the value of Δ for Δ-stepping, blocking size of EdgeBlocking, and thresholds for hybrid schedules need to be appropriately selected for each input graph and algorithm. Searching through the huge optimization space exhaustively is very time-consuming.

To navigate the schedule space more efficiently, we built an autotuner using OpenTuner [4]. For each direction, the autotuner chooses among all 288 combinations of options for load balancing, deduplication, output frontier strategy, blocking, traversal direction, and kernel fusion. For direction-optimized schedules that involve two traversal directions, the autotuner combines together two sets of schedules, one for each direction. The autotuner converges within 10 minutes on each input graph for most algorithms and produces a schedule that matches the performance of hand-optimized schedules.

THIS PAGE IS INTENTIONALLY LEFT BLANK

Chapter 6

Evaluation

We compare the performance of the code generated from UGF's GPU backend with other state-of-the-art GPU graph frameworks and libraries on five graph algorithms and nine different graph inputs. We also study the performance impact of EdgeBlocking and ETWC. The evaluation for GPUs is performed on both an NVIDIA Titan Xp (Pascal-generation GPU with 12GB of GDDR5 main memory, 3MB of L2 cache, and 48KB of L1 cache per SM with a total of 30 SMs) and an NVIDIA Tesla V100 (Volta-generation GPU with 32 GB of HBM2 main memory, 6MB of L2 cache, and 128KB of combined L1 cache and shared memory per SM with a total of 80 SMs). For evaluation of the CPU GraphVM, we run the experiments on a dual-socket system with Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads. The machine has 128 GB of DDR3-1600 memory and 30 MB last level cache on each socket and runs with Transparent Huge Pages (THP) enabled. Finally, for evaluating Swarm, we simulate systems with up to 256 simple in-order cores, with 64 KB/core of L2 cache and 256 KB/core of L3 cache (for 64 MB total L3 capacity on the 256-core chip). Simulation parameters and methodology are the same as in prior work [29, 63, 32].

Data sets. We list the input graphs used for our evaluation in Table 6.1 along with the sources they are downloaded from. Out of the 9 graphs, OK, TW, LJ, SW, HW, and IC have power-law degree distributions while RU, RN, and RC are road graphs with bounded degree distributions and high diameter.

Graph Input	Vertex Count	Directed Edge Count
Orkut (OK) [55]	2,997,166	212,698,418
Twitter (TW) [55]	21,297,772	530,051,090
LiveJournal (LJ) [17]	4,847,571	85,702,474
Sinaweibo (SW) [55]	58,655,849	522,642,066
Hollywood (SW) [17]	1,139,905	112,751,422
Indochina (IC) [17]	7,414,865	301,969,638
RoadUSA (RU) [17]	23,947,347	57,708,624
RoadCent (RC) [17]	14,081,816	33,866,826
RoadCA (RN) [17]	1,971,281	5,533,214

Table 6.1: Graph inputs used for evaluation.

Algorithms. We evaluate frameworks using PageRank (PR), Breath-First Search (BFS), DeltaStepping for Single-Source Shortest Paths (SSSP), Connected Components (CC), and Betweenness Centrality (BC). PageRank is a topology-driven algorithm where we iterate over all the edges in every round. BFS generates the parent tree. SSSP uses the DeltaStepping algorithm. CC, BC are all data-driven algorithms where we iterate over only a subset of the edges.

Existing Frameworks. We compare UGF's performance with three state-of-the-art GPU graph processing frameworks: Gunrock [68], GSwitch [42], and SEPGraph [66]. All of the frameworks have optimized implementations of BFS for power-law graphs using direction-optimization [6]. SEPGraph improves the performance of DeltaStepping and BFS on high-diameter graphs with asynchronous execution. GSwitch chooses among several optimal parameters for traversal direction, load balancing, and frontier creation using a learned decision tree that makes use of graph characteristics and runtime metrics. These frameworks implement all of the algorithms that we evaluate, except for CC, PageRank, and BC, which are not implemented in SEPGraph. SEPGraph instead implements PageRank-Delta which is a data-driven algorithm that iterates over only a subset of edges and vertices. Since PageRank-Delta performs less work per iteration, we cannot compare the performance against PageRank.

	PageRank (time per round)			CC			BC		
Graph	UGF	GU	GW	UGF	GU	GW	UGF	GU	GW
OK	**14.18**	60.63	117.10	**63.31**	71.95	76.85	**26.22**	213.93	37.93
TW	**77.86**	113.55	211.03	**196.78**	374.59	OOM	174.83	505.31	**122.81**
LJ	**7.68**	17.67	OOM	**24.65**	35.96	27.81	28.93	88.04	**26.98**
SW	**102.11**	178.70	338.79	**276.04**	439.51	OOM	**204.32**	1095.60	415.06
HW	**7.01**	22.67	OOM	**12.04**	37.43	18.23	**12.20**	29.03	79.44
IC	18.24	13.16	**9.30**	**31.66**	235.92	43.10	35.78	47.97	**10.70**
RU	**6.32**	10.53	7.62	**20.66**	74.45	31.21	**302.22**	987.93	564.53
RC	**5.56**	9.96	8.86	**27.03**	48.22	27.13	**239.55**	632.97	332.91
RN	**0.43**	0.94	0.47	**1.72**	5.82	3.04	**24.05**	86.44	39.51
	SSSP with DeltaStepping				BFS				
Graph	UGF	GU	GW	SEP-G	UGF	GU	GW	SEP-G	-
OK	**94.30**	978.44	550.38	434.77	**1.75**	1.92	1.94	6.40	
TW	**114.34**	264.54	233.54	237.12	**21.13**	OOM	22.52	40.18	
LJ	**54.46**	260.19	172.99	172.07	4.66	4.80	**3.68**	11.63	
SW	**685.66**	3470.59	1933.29	2296.01	20.17	OOM	**18.96**	93.69	
HW	**18.44**	74.90	47.54	92.92	**1.89**	2.04	2.30	4.76	
IC	**120.42**	232.55	268.95	511.70	**10.68**	15.92	70.69	55.15	
RU	601.08	53518.87	1238.03	**440.21**	**73.15**	442.19	119.93	84.29	
RC	337.62	29443.51	642.38	**286.03**	**49.59**	OOM	84.98	61.51	
RN	17.01	89.18	27.80	**16.48**	**6.13**	OOM	10.29	6.88	

Table 6.2: Execution time in milliseconds for the five algorithms on 9 input graphs for the 4 frameworks in comparison, UGF (Universal Graph Framework), Gunrock, GSwitch, and SEP-G (SEPGraph) running on an NVIDIA Titan Xp GPU. The fastest results for each algorithm-graph input are marked in **bold**. The PageRank, BFS, BC, and CC algorithms use unweighted and symmetrized graphs. Single-Source Shortest Path (SSSP) with DeltaStepping uses unsymmetrized graphs with edge weights. Uniformly random integer weights between 1–1000 are added for Orkut, Sinaweibo, Hollywood, and Indochina because they did not have weights originally. OOM indicates that the framework ran out of memory for the particular input and – indicates that the framework does not implement the algorithm.

	PageRank (time per round)			CC			BC		
Graph	UGF	GU	GW	UGF	GU	GW	UGF	GU	GW
OK	**10.87**	20.59	32.77	**15.18**	-	16.33	**20.11**	213.93	20.56
TW	**35.83**	43.04	103.88	**136.73**	-	137.12	94.93	505.31	**56.44**
LJ	**4.54**	7.66	6.49	**7.96**	-	11.12	20.12	88.04	**16.88**
SW	**56.23**	66.41	163.88	**174.91**	-	290.02	392.24	1095.60	**216.05**
HW	**4.15**	6.47	8.67	8.23	-	**6.85**	**6.04**	29.03	6.08
IC	**13.68**	16.64	46.55	22.08	-	**4.46**	**10.67**	47.97	55.58
RU	3.14	**2.68**	3.22	**12.85**	-	17.94	**395.26**	987.93	536.95
RC	**2.17**	2.61	2.97	**10.66**	-	14.14	**247.8**	632.97	336.78
RN	**0.23**	0.27	0.29	**1.42**	-	2.79	**40.43**	86.44	47.06
	SSSP with DeltaStepping				BFS				
Graph	UGF	GU	GW	SEP-G	UGF	GU	GW	SEP-G	-
OK	**53.47**	243.05	199.59	164.69	**1.51**	1.66	**1.51**	5.72	
TW	**62.10**	97.52	132.94	117.97	13.60	13.52	**10.60**	38.18	
LJ	42.85	**14.55**	77.95	103.40	**2.56**	3.63	3.05	9.59	
SW	645.26	**235.75**	1062.56	1066.57	83.74	94.77	**12.26**	70.70	
HW	16.21	**5.87**	26.77	51.75	1.63	1.67	**1.60**	4.81	
IC	155.09	**13.63**	211.85	350.58	31.04	**12.57**	40.60	39.39	
RU	253.25	788.23	390.23	**191.08**	**74.53**	775.16	186.43	97.62	
RC	195.40	429.24	222.05	**128.02**	119.66	434.01	**115.13**	65.49	
RN	26.78	66.79	32.47	**19.46**	10.25	68.35	16.76	**9.33**	

Table 6.3: Execution time in milliseconds for the same experiments in Table 6.2 running on an NVIDIA V100 GPU.

6.1 Comparison with Existing Frameworks on a Pascal GPU

Table 6.2 shows the execution times of all of the algorithms in UGF and the other frameworks on a Pascal GPU. UGF outperforms the next fastest of the three frameworks on 36 out of 45 experiments by up to 5.1× and is never more than 36% slower than the fastest framework on the rest of the experiments. Table 6.8 shows UGF always uses significantly fewer lines of code compared to other frameworks. We show the best schedules selected for all the applications and graphs in the supplementary materials.

PageRank. UGF has the fastest PageRank on 8 out of 9 graphs. Compared to Gunrock and GSwitch (SEPGraph does not provide the original PageRank algorithm), UGF is up to 4.2x faster. This is mainly because of the EdgeBlocking optimization that reduces the number of L2 cache misses, as described in Section 5.2. For Hollywood and Indochina, the graphs are already clustered, meaning that each vertex only has neighbors in a small range of vertices.

BFS. UGF has the fastest BFS on 7 of the 9 graphs. UGF outperforms GSwitch and Gunrock by up to 6.04x and 1.63x, respectively, on the road graphs because Gunrock and GSwitch do not use the Kernel Fusion optimization which reduces kernel launch overheads as discussed in Section 2.5. SEPGraph is only up to 1.24x slower than UGF on the road graphs because it uses asynchronous execution. However, the better load balancing achieved by using ETWC makes UGF faster than SEPGraph. On the power-law graphs, direction optimization is very effective. Both Gunrock and GSwitch use direction optimization and hence the performance of UGF is very close to both of them. GSwitch and Gunrock also use idempotent label updates, which eliminates atomic compare-and-swap operations.

Indochina is a special case of a power-law graph that does not benefit from direction optimization because the number of active vertices increases very gradually. Thus for most of the iterations, the PUSH strategy is the most efficient. Both Gunrock and

GSwitch use direction optimization for Indochina and suffer from the extra work done in the PULL direction. UGF uses the more efficient (PUSH only) schedule.

CC. For CC, UGF is the fastest on all the graph inputs. UGF is up to 3.4x faster than the next fastest framework. All of the frameworks use the same algorithm and execution strategies. We tune the performance by choosing different load balancing strategies for each graph (ETWC for power-law graphs and CM for road-graphs). Table 6.6 shows the impact of choosing an efficient load balancing strategy for the CC algorithm.

DeltaStepping. UGF has the fastest DeltaStepping performance on 6 out of the 9 graph inputs and runs up to 5.11x faster than the next fastest framework. DeltaStepping needs the Kernel Fusion optimization for road graphs because of their high diameter and low number of vertices processed in each iteration. Without this optimization, GSwitch and Gunrock are slower on road graphs by up to 2.05x and 89x, respectively. On power-law graphs, UGF benefits from the better ETWC load balancing strategy and performs up to 5.11x faster than the next fastest framework. On road graphs, SEPGraph executes up to 1.36x times faster because of the highly optimized asynchronous execution. Table 6.4 shows that UGF's execution time on CPU is up to 2x faster than that of SEPGraph.

BC. UGF has the fastest BC performance on 8 out of the 9 graph inputs. UGF run up to 2.03x faster than the next fastest framework and is never more than 1.4x slower. Gunrock does not use direction optimization, which is critical for high performance on power-law graphs. GSwitch uses direction optimization, but UGF outperforms GSwitch because of the better load balancing from ETWC. For high diameter graphs, UGF benefits greatly from the Kernel Fusion optimization. Both GSwitch and Gunrock do not implement Kernel Fusion.

6.2 Comparison with Existing Frameworks on a Tesla V100 GPU

To demonstrate UGF's portability across different GPU architectures, we also compare the performance of UGF against existing frameworks on an NVIDIA V100 GPU as shown in Table 6.3. The Volta-generation GPU requires different code generation because of different hardware features, such as a faster and larger L1 cache and threads not being required to execute in lockstep.

Similar to the Pascal experiments, we ran the five algorithms—PageRank, SSSP, CC, BFS, and BC—on 9 graphs and compared the performance of UGF with the three other frameworks. Currently, the CC algorithm is explicitly disabled on the Gunrock repository because their implementation does not support the newer GPUs. Thus we have not included the performance. Table 6.3 shows the execution times in milliseconds for all of the experiments. UGF is the fastest for 26 out of 45 experiments with up to 4.49× speedup over the next fastest framework. For most of the algorithms and graphs, UGF follows similar trends as that of the Pascal experiments. Just like for the Pascal GPU, we also apply the EdgeBlocking optimization for PageRank. But due to the larger L2 cache size on the V100 GPU, each sub-graph can hold more vertices and thus there is less overhead from duplicating vertices.

UGF is still faster than GSwitch and Gunrock on data-driven algorithms like SSSP, CC, and BFS for high diameter graphs because UGF implements the kernel fusion optimization. We see that the speedup for CC on power-law graphs is not as significant as in Pascal because V100 has more SMs and semi-warp execution, and our ETWC load balancing scheme is less effective. Finally, in some cases for algorithms like SSSP, the execution times for all frameworks on Volta is higher than that of Pascal. This is because the kernel launch overhead and thread synchronization costs for Volta are higher than that of Pascal. And since applications with limited available parallelism are not able to saturate the higher number of threads on Volta, these costs start slowing down the application.

SSSP	UGF CPU	UGF GPU	SEPGraph
RU	**212.30**	601.80	440.21
RC	**162.49**	337.62	286.03
OK	106.00	**94.30**	434.77
LJ	90.05	**54.46**	172.07

Table 6.4: Comparisons of UGF-generated CPU, UGF-generated GPU, and SEPGraph implementations on SSSP with DeltaStepping. The running times are in milliseconds. We do not count the data transfer time from CPU to GPU.

6.3 Comparison against CPU

We compare the performance of UGF-generated CPU implementations with GPU implementations. We ran the evaluation on a dual-socket system as specified earlier. UGF generates the same CPU implementation as in the previous GraphIt framework [74, 76].

On PageRank, BFS, and CC, the GPU implementations are faster because these algorithms can easily utilize the large amount of parallelization and memory bandwidth available on the GPUs. On the other hand, DeltaStepping, which has less parallelism available when running on road graphs, executes up to 2.07× faster on the CPU due to more powerful cores and larger caches. The execution times for DeltaStepping on GPU and CPU can be found in Table 6.4. Furthermore, GPUs cannot even process some graphs that are much larger than the GPU memory but run fine on CPUs.

These experiments provide evidence for our claim that a single hardware is not suitable for all algorithms and shows how a unified framework that decouples algorithms and optimizations from the target hardware can help users achieve high performance.

Graph	ETWC	TWC	CM
OK	43.58	**40.69**	42.24
TW	**106.11**	107.57	116.06
LJ	19.72	20.03	**18.42**
SW	**226.35**	230.00	230.03
HW	**4.94**	5.79	8.17
IC	**11.38**	11.50	22.16
RU	**136.64**	255.89	168.90
RC	**91.20**	162.54	109.89
RN	**13.10**	25.77	16.25

Table 6.5: Execution time (in milliseconds) of BFS (PUSH only) using ETWC, TWC, and CM load-balancing strategies. The fastest results are in **bold**.

Graph	ETWC	TWC	CM
OK	69.93	81.10	**63.31**
TW	**196.78**	232.58	252.83
LJ	28.87	31.64	**24.65**
SW	431.02	**276.04**	379.12
HW	**12.04**	12.80	14.76
IC	42.06	**31.66**	42.94
RU	32.33	24.73	**20.66**
RC	28.42	37.62	**27.03**
RN	3.02	1.92	**1.72**

Table 6.6: Execution time (in milliseconds) of CC using the ETWC, TWC, and CM load-balancing strategies. The fastest results are in **bold**.

Graph	Without EB	With EB	Speed up	Preprocess time
OK	41.75	**14.18**	2.94x	22.75
TW	88.25	**77.86**	1.13x	129.43
LJ	15.67	**7.68**	2.04x	20.65
SW	144.88	**102.11**	1.41x	141.86
HW	**7.01**	7.02	0.99x	12.00
IC	**18.24**	19.55	0.93x	32.25
RU	8.35	**6.32**	1.35x	11.77
RC	8.39	**5.56**	1.50x	8.20
RN	0.44	**0.43**	1.02x	1.08

Table 6.7: Execution time (in milliseconds) per round of PageRank with and without Edge-Blocking. The fastest results are in **bold**.

Framework	PR	BFS	SSSP	CC	BC
Gunrock	2207	2189	1438	3014	1792
GSwitch	159	164	203	160	280
SEPGraph	–	481	473	–	–
UGF	**61**	**66**	**50**	**62**	**128**

Table 6.8: Number of lines of code for the five algorithms written using Gunrock, GSwitch, SEPGraph, and UGF. SEPGraph does not implement CC, PageRank and BC. The fewest number of lines is in **bold**. The number of lines for UGF includes both the algorithm and the schedule.

6.4 Performance of ETWC and EdgeBlocking

We evaluate the performance of the two new optimizations, ETWC and EdgeBlocking using BFS and PageRank. To evaluate the ETWC load-balancing scheme, we run the BFS algorithm on all nine graph inputs. For these experiments we used only the PUSH strategy and disabled the Kernel Fusion optimization. The rest of the scheduling parameters are fixed to the best values we found for each graph. We then vary the load balancing scheme to be ETWC, TWC, and CM (TWC and CM are the best performing methods on power-law graphs and road graphs , respectively, and hence other load-balancing schemes are not shown). The results are shown in Table 6.5. CM is faster than TWC on graphs that have a regular degree distribution, such as road graphs and the Indochina graph, while TWC performs better on power-law graphs. This is because TWC can adapt to large variation in degrees of vertices in the power-law graphs, but at the same time is slower on road graphs due to the overhead from load balancing. ETWC outperforms the other two load-balancing schemes on 7 of the 9 graphs. ETWC does well both on power-law graphs and road graphs because it is able to achieve good load balancing without incurring a large overhead. Unlike TWC, ETWC balances the edges only locally within a CTA, thus communicating only using shared memory. We show results for similar experiments with CC in Table 6.6. In the case of CC, even though ETWC is not always the fastest scheme, it is close to the fastest for most graphs.

We evaluate EdgeBlocking by running PageRank on all of the input graphs, and

	SSSP					PageRank (time per round)				
Graph	1	4	16	64	256	1	4	16	64	256
RN	1.569	0.229	0.057	0.034	0.022	0.355	0.092	0.029	0.010	0.005
RC	11.591	1.924	0.442	0.228	0.143	3.562	0.906	0.277	0.077	0.036
RU	18.752	2.926	0.673	0.387	0.246	4.386	1.143	0.357	0.119	0.057
HW	26.078	5.671	1.757	0.680	0.407	4.899	1.187	0.364	0.125	0.243
LJ	30.775	7.032	2.028	0.650	0.332	5.999	1.428	0.403	0.121	0.082
OK	86.424	20.184	5.410	1.430	0.691	17.153	3.948	1.049	0.282	0.125

Table 6.9: Execution time (in seconds) of the code generated by the Swarm GraphVM for the SSSP and PageRank (time per round) application with varying number of cores.

the results are shown in Table 6.7. We fix the schedule to use edge-only load-balancing (which is the fastest for PageRank) and compare the execution times with and without EdgeBlocking. The vertex data of these vertices fit in the L2 cache of the GPU. PageRank runs up to 2.94x faster with EdgeBlocking enabled. EdgeBlocking causes some slowdown on Indochina and Hollywood because of lower work-efficiency. Table 6.7 also shows the preprocessing time for each of the input graphs. We can see that the preprocessing time for all the graphs is less than three iterations of PageRank and thus is easily amortized.

6.5 Evaluation of Swarm GraphVM

Finally, we evaluate the Swarm GraphVM by running the code it generates on the Swarm hardware simulator. When tuning the scheduling parameters, we find that Swarm offers unique performance characteristics. For example, SSSP with DeltaStepping obtains the best performance with Δ set to 1, as Swarm readily extracts parallelism across priority levels using speculative execution. UGF's scheduling language makes it easy to tune each algorithm for different architectures.

We vary the number of cores to measure the scalability of the generated code. Scalability is an important metric because the Swarm hardware uses in-order cores that are simpler than conventional CPU cores, and relies on massive parallelism to achieve good performance. Table 6.9 shows the execution time in seconds for the PageRank and the SSSP with DeltaStepping applications. We can see that with 256 cores, we are able to get up to 95x speedup over a single core. This speedup shows that the generated code is able to utilize the Swarm hardware effectively to extract speculative parallelism available in the program. Performance improves as the core

count increases to hundreds of cores, unlike conventional multi-core systems where performance for SSSP saturates or even decreases with increasing the number of cores.

6.6 Feature comparisons against other frameworks

Graph Frameworks for CPUs. There has been a large body of work on graph processing for shared-memory (e.g., [60, 61, 21, 27, 20, 75]), distributed-memory (e.g., [13, 77, 15, 41]), and external-memory (e.g., [36, 56, 65, 67, 73, 79, 40, 78]). These frameworks support a limited set of optimizations and cannot achieve consistent high performance across different algorithms and graphs. Abelian [20] uses the Galois framework as an interface for shared-memory CPU, distributed-memory CPU, and GPU platforms. However, it lacks support for direction-optimization, various load balancing optimizations, and active vertex set creation optimizations, which are needed to achieve high performance. GraphIt [76, 74] is a domain-specific language that expands the optimization space to outperform other CPU frameworks by decoupling algorithm from optimizations. However, GraphIt generates code only for CPUs. UGF extends GraphIt by decoupling algorithms, optimizations, and hardware with the GraphIR to enable efficient implementations for CPUs, GPUs, and potentially other architectures by exploring a large optimization space (Table 6.10).

Graph Frameworks for GPUs. There has been tremendous efforts on developing high-performance graph processing frameworks on GPUs (e.g., [43, 52, 9, 66, 39, 50, 34, 33, 68, 24, 28, 38, 59, 25, 16, 62, 49, 12, 35, 19, 23]). Gunrock [68] proposes a novel data-centric abstraction and incorporates existing GPU optimization strategies. IrGL [52] introduces the Kernel Fusion optimization that reduces the overhead of kernel launch for SSSP and BFS on high-diameter road networks. However, most of the frameworks support only a subset of existing optimizations and cannot achieve high performance on all algorithms and graphs using GPUs [42]. GSWITCH [42] identifies and implements a set of useful optimizations and uses autotuning to achieve high performance.

Compared to the existing GPU frameworks, UGF further expands the optimization space by an order of magnitude by decoupling the algorithm from GPU performance

Framework	**UGF**	GSwitch	SEPGraph	Gunrock
Load Balancing	**6**	4	1	3
Edge Blocking	**2**	1	1	1
Vertexset Creation	**3**	**2**	1	**2**
Kernel Fusion	**2**	1	**2**	1
Direction Optimization	**2**	**2**	**2**	**2**
Vertexset Deduplication	**2**	1	**2**	**2**
Vertices Ordering	**2**	**2**	**2**	**2**
Total Schedules (One Direction)	**288**	16	8	24

Table 6.10: Number of options in each category of optimizations in the different GPU graph frameworks. The last row shows the total number of possible schedules supported by the framework for one direction

optimizations and introducing new optimizations, ETWC and EdgeBlocking, as shown in Table 6.10. UGF's scheduling language and compiler enable us to find new combinations of optimizations, which gives up to 5.1× speedup over the fastest of state-of-the-art GPU frameworks. UGF also supports generating code for both CPU and GPU platforms without changing the algorithm specification.

Domain-Specific Accelerators (DSAs) for Graph Processing. DSAs can enhance the performance and energy-efficiency for graph processing [45, 46, 22, 30, 57, 37, 14, 2, 69, 3]. Swarm [31, 32, 63] employs hardware-accelerated speculative execution to achieve high parallelism and good scalability. Tesseract [3] utilizes in-memory graph processing to reduce the overhead for memory transactions. Graphicionado [22] enables energy-efficient processing while achieving high performance with memory subsystem specialization. One of the main goals of UGF is to make it easy to develop backends for these different DSAs.

Load-balancing and Locality-enhancing Optimizations on GPUs. TWC is a dynamic load balancing strategy designed for efficient BFS on GPUs [43]. Gunrock [68] and GSwitch [42] implement both TWC and a few other static load balancing techniques, which we describe in more detail in Section 2.5. The irregular memory access pattern in graph algorithms makes it hard to take advantage of the memory hierarchy [7, 75]. To reduce the irregular memory access, several tiling approaches for graphs have been proposed [75, 72, 77, 64]. However, a naive graph tiling approach on GPU results in poor performance due to insufficient parallelism. EdgeBlocking finds a good balance between locality and parallelism by tiling for the shared L2 cache on GPUs and processing one tiled subgraph at a time. Previous work [48, 26] have

employed tiling strategies for sparse matrix operations.

Scheduling Languages. Scheduling languages have been proposed in several other domains. Halide [54] decouples algorithms with scheduling for image processing pipelines, which allows users to explore optimization space without modifying the algorithmic code. Halide also explores the tradeoffs between locality, parallelism, and redundant computations. PolyMage [47] and Tiramisu [5] also support a scheduling language based on a polyhedral model. However, these scheduling languages only support loop optimizations for dense arrays. GraphIt [76] also adopts the concept of decoupling graph algorithms with scheduling. However, GraphIt only supports graph algorithms on CPUs, while our approach can be applied across different architectures such as GPU and Swarm.

Chapter 7

Conclusions

7.1 Summary

This thesis introduces Universal Graph Framework (UGF), a novel graph processing framework for writing high-performance graph applications for GPUs, CPUs, and Swarm. UGF decouples algorithm, schedule, and the hardware with a novel intermediate representation, the GraphIR. The GraphIR encodes information for algorithms and hardware-independent optimizations, enabling the reuse of high-level program analyses and transformations across different backends. The extensible GPU scheduling language and compiler enables users to search through many different combinations of load-balancing, traversal direction, active vertexset management, and work-efficiency optimizations. This thesis also proposes two new performance optimizations, ETWC and EdgeBlocking, to improve load-balancing and locality of edge processing on GPUs. We evaluated UGF on five algorithms and 9 graphs and showed that it achieves up to 5.1× speedup over the next fastest state-of-the-art GPU framework, and is the fastest in 62 out of 90 experiments on GPUs. Similarly, UGF generates CPU implementations that match the performance of the original GraphIt compiler and also generates efficient code for the Swarm architecture.

7.2 Future directions

This thesis presents UGF a framework that enables graph experts to generate code for different hardware backends with easy. The GraphIR and the extensible scheduling language allows the developer to only have to deal with hardware specific optimizations while reusing the hard-independent optimizations in the compiler.

Recently, many new hardware platforms have been proposed targetting graph and other sparse applications including the Symphony architecture from NVIDIA or the Hammerblade architecture. All these architectures have very different performance characteristics which present a very different optimization space from traditional CPUs and GPUs. These architectures also provide a very different programming model. For example, the Symphony architecture is a dataflow architecture and the Hammerblade architecture with its large number of cores and high-bandwidth memory transfers is very different to program from CPUs.

UGF can be used to rapidly target different graph applications to run on these upcoming architectures from the same high-level algorithmic description. This will help both the application writers to choose the best platform for their applications and inputs and also help the architects understand the characteristics of these applications and better adapt their hardware to suit the needs.

We are planning to work with other groups and build GraphVMs to target these architectures. As a part of this work, we will also explore the optimization space for these architectures and use the extensible scheduling language to create a scheduling language specifically tailored for them.

Appendix A

Optimum GG schedules

Table A.1 shows the optimum schedule selected for each graph input and algorithm for the GPU experiments.

Algo	Power-Law graphs	Road Graphs

BFS	`SimpleGPUSchedule s1;`
	`s1.configDeduplication(DISABLED);`
	`s1.configLoadBalance(ETWC);`
	`s1.configDirection(PUSH);`
	`s1.configFrontierCreation(FUSED);`
	`SimpleGPUSchedule s2 = s1;`
	`s2.configLoadBalance(VERTEX_BASED);`
	`s2.configDirection(PULL, BITMAP);`
	`s2.configDeduplication(DISABLED);`
	`s2.configFrontierCreation(UNFUSED_BITMAP);`
	`HybridGPUSchedule h1`
	`(INPUT_VERTEXSET_SIZE, "argv[3]", s1, s2);`
	`program->applyGPUSchedule("s0:s1", h1);`
		`SimpleGPUSchedule s1;`
		`s1.configDeduplication(DISABLED);`
		`s1.configLoadBalance(ETWC);`
		`s1.configDirection(PUSH);`
		`s1.configFrontierCreation(FUSED);`
		`program->applyGPUSchedule("s0:s1", s1);`
		`SimpleGPUSchedule s0;`
		`s0.configKernelFusion(ENABLED);`
		`program->applyGPUSchedule("s0", s0);`

PR	SimpleGPUSchedule s1;
	s1.configDirection(PULL);
	s1.configLoadBalance(EDGE_ONLY, BLOCKED,
	0x42000);
	program->applyGPUSchedule("s1", s1);
		SimpleGPUSchedule s1;
		s1.configDirection(PULL);
		s1.configLoadBalance(EDGE_ONLY, BLOCKED, 0x42000);
		program->applyGPUSchedule("s1", s1);

SSSP	SimpleGPUSchedule s1;
	s1.configLoadBalance(ETWC);
	s1.configFrontierCreation(UNFUSED_BOOLMAP);
	s1.configDelta("argv[3]");
	program->applyGPUSchedule("s0:s1", s1);
		SimpleGPUSchedule s1;
		s1.configLoadBalance(ETWC);
		s1.configFrontierCreation(FUSED);
		s1.configDelta("argv[3]");
		program->applyGPUSchedule("s0:s1", s1);
		SimpleGPUSchedule s0;
		s0.configKernelFusion(ENABLED);
		program->applyGPUSchedule("s0", s0);

CC	SimpleGPUSchedule s1;
	s1.configLoadBalance(ETWC);
	s1.configDeduplication(ENABLED);
	s1.configFrontierCreation(UNFUSED_BITMAP);
	program->applyGPUSchedule("s1", s1);
		SimpleGPUSchedule s1;
		s1.configLoadBalance(ETWC);
		s1.configDeduplication(ENABLED);
		s1.configFrontierCreation(UNFUSED_BITMAP);
		program->applyGPUSchedule("s1", s1);

BC	SimpleGPUSchedule s1;
	s1.configLoadBalance(ETWC);
	s1.configFrontierCreation(FUSED);
	s1.configDeduplication(ENABLED, FUSED);
	SimpleGPUSchedule s2;
	s2.configLoadBalance(ETWC);
	s2.configDirection(PULL, BITMAP);
	s2.configFrontierCreation(UNFUSED_BITMAP);
	HybridGPUSchedule h1
	(INPUT_VERTEXSET_SIZE, "argv[3]", s1, s2);
	program->applyGPUSchedule("s1", s1);
	program->applyGPUSchedule("s2", s1);
		SimpleGPUSchedule s1;
		s1.configLoadBalance(ETWC);
		s1.configFrontierCreation(FUSED);
		s1.configDeduplication(ENABLED, FUSED);
		program->applyGPUSchedule("s0:s1", s1);
		program->applyGPUSchedule("s2:s3", s1);
		SimpleGPUSchedule s0;
		s0.configKernelFusion(ENABLED);
		program->applyGPUSchedule("s0", s0);
		program->applyGPUSchedule("s2", s0);

Table A.1: The schedules we used for the five applications. The table shows separate schedules for power-law graphs and road-graphs. The delta and threshold parameters are input at runtime and are different for each graph. Hence they are not shown in this table.

Bibliography

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. volume 42, pages 20:1–20:44, October 2017.

[2] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145, Sep. 2018.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[6] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 12:1–12:10, 2012.

[7] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65, 2015.

[8] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In

Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 235–248, 2017.

[10] Maciej Besta, Michal Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, 2017.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 49–60, 2013.

[12] Shuai Che. Gascl: A vertex-centric graph model for gpus. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[13] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *TOPC*, 5(3):13:1–13:39, 2018.

[14] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, April 2019.

[15] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 752–768, 2018.

[16] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[17] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[18] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 World Wide Web Conference (WWW)*, pages 1775–1784, 2018.

[19] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 121–131, 2019.

[20] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Euro-Par*, pages 249–264, 2018.

[21] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 246–260, 2018.

[22] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[23] 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)*, pages 233–245, 2017.

[24] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing (HiPC)*, pages 197–208, 2007.

[25] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. Multigraph: Efficient graph processing on gpus. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40, 2017.

[26] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 300–314, 2019.

[27] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–362, 2012.

[28] 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)*, pages 267–276, 2011.

[29] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-centric execution of speculative parallel programs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[30] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[31] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.

[32] Mark C. Jeffrey, Victor A. Ying, Suvinay Subramanian, Hyun Ryong Lee, Joel Emer, and Daniel Sanchez. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 217–230, Oct 2018.

[33] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, Oct 2015.

[34] 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*, pages 239–252, 2014.

[35] 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 ACM SIGMOD International Conference on Management of Data*, pages 447–461, 2016.

[36] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[37] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. GraphIA: An in-situ accelerator for large-scale graph processing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 79–84, 2018.

[38] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[39] Hang Liu and H Howie Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *USENIX Annual Technical Conference*, pages 411–428, 2019.

[40] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543, 2017.

[41] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[42] 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)*, pages 201–213, 2019.

[43] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. volume 1, pages 14:1–14:30, February 2015.

[44] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[45] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2018.

[46] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1022, 2019.

[47] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News*, 43(1):429–443, 2015.

[48] Y. Nagasaka, A. Nukada, and S. Matsuoka. Cache-aware sparse matrix formats for kepler gpu. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 281–288, Dec 2014.

[49] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 463–474, 2013.

[50] 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)*, pages 622–636, 2018.

[51] NVIDIA. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, August 2019.

[52] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 2016.

[53] Stefano Pallottino and Maria Grazia Scutellà. *Shortest Path Algorithms In Transportation Models: Classical and Innovative Aspects*, pages 245–281. 1998.

[54] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[55] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[56] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013.

[57] Albert Segura, Jose-Maria Arnau, and Antonio González. SCU: A GPU stream compaction unit for graph processing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 424–435, 2019.

[58] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: Real-time content recommendations at twitter. *Proc. VLDB Endow.*, 9(13):1281–1292, September 2016.

[59] Xuanhua Shi, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, Bingsheng He, and Hai Jin. Frog: Asynchronous graph processing on gpu with hybrid coloring model. *IEEE Transactions on Knowledge and Data Engineering*, 30(1):29–42, 2017.

[60] 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)*, pages 135–146, 2013.

[61] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *IEEE Data Compression Conference (DCC)*, pages 403–412, 2015.

[62] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A fast gpu algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[63] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proceedings of the 44th Annual*

International Symposium on Computer Architecture (ISCA), pages 587–599, June 2017.

[64] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 16:1–16:10, 2017.

[65] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference*, pages 507–522, 2016.

[66] 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)*, pages 38–52, 2019.

[67] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single PC. In *USENIX Annual Technical Conference*, pages 387–401, 2015.

[68] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3, 2017.

[69] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 8:1–8:12, 2018.

[70] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 974–983, 2018.

[71] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. T4: Compiling sequential code for effective speculative parallelization in hardware. In *Proceedings of the 47th International Symposium in Computer Architecture (ISCA)*, June 2020.

[72] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 183–193, 2015.

[73] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. pages 608–621, 2018.

[74] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158âĂŞ170, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, 2017.

[76] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.

[77] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 301–316, 2016.

[78] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference*, pages 375–386, 2015.

[79] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the Fourteenth EuroSys Conference*, page 38, 2019.