Unified Graph Framework: Optimizing Graph Applications across Novel Architectures

by

Claire Hsu

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2020

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
June 4, 2021

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

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Unified Graph Framework: Optimizing Graph Applications across Novel Architectures

by

Claire Hsu

Abstract

High performance graph applications are crucial in a wide set of domains, but their performance depends heavily on input graph structure, algorithm, and target hardware. Programmers must develop a series of optimizations either on the compiler level, implementing different load balancing or edge traversal strategies, or on the architectural level, creating novel domain-specific accelerators optimized for graphs. In recent years, there has been rapid growth on the architectural end, with each novel architecture contributing new potential optimizations.

To help compiler development scale with the growth of the architecture domain, we develop the Unified Graph Framework (UGF) to achieve portability for easy integration of novel backend architectures into a high-level hardware-independent compiler. UGF builds on the GraphIt domain-specific language, which divides algorithm specification from scheduling optimizations, and separates hardware-independent from hardware-specific scheduling parameters and compiler passes. As part of UGF, we introduce GraphIR, a graph-specific, hardware-independent intermediate representation; an extensible scheduling language API that enables hardware-independent optimizations and programmer-defined hardware-specific optimizations; and the GraphVM, the compiler backend implemented for each hardware architecture.

Lastly, we evaluate UGF by implementing a GraphVM for Swarm, a recently developed multicore architecture. We integrate several scheduling optimizations built around Swarm's ability to speculate on fine-grained tasks in future loop iterations. When evaluated on five applications and 10 input graphs, UGF successfully generates highly optimized Swarm code and achieves up to 8x speedup over baseline implementations.

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

Acknowledgments

I'd like to thank my thesis advisor Professor Saman Amarasinghe for guiding me through the research process, starting as a senior in the SuperUROP program to an MEng student this year. He taught me the importance of asking good questions, and his excitement about all sorts of research problems always inspired me.

I would like to thank everyone on the GraphIt project for advising me over the years - Yunming Zhang, Ajay Brahmakshatriya, Changwan Hong, Shoaib Kamil, and Professor Julian Shun. Special thanks to Yunming, one of my mentors for the past two years, who introduced me to the GraphIt project and patiently taught me so much about performance engineering and benchmarking. I'd also like to especially thank Ajay for his incredible mentorship and guidance during my MEng year, from helping me with presentations to thesis writing, to helping me structure my MEng project and debug code. Finally, thank you to the entire COMMIT group for your wisdom and enthusiasm. I have learned an incredible amount over the past two years, and am extremely grateful for the opportunity to collaborate with and learn from you all.

Thank you to our collaborators on our ISCA 2021 paper - I'd like to express my gratitude to Victor Ying for helping me with the Swarm architecture and my thesis project, and to Emily Furst for guidance on the Hammerblade architecture.

Thank you to my family - mom, dad, and grandparents - for believing in me since day 1 and encouraging me to pursue my interests. I am so grateful for everything you've done for me, and for teaching me to dream big and never give up. Lastly, thank you to my good friends, Karunya, Yara, and Afeefah, for your unwavering support during my time at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph applications have long been essential in a broad range of domains, such as machine learning algorithms, social media network analytics [25, 17], and protein structure modeling [36, 16]. These applications also operate on huge datasets, often involving graphs with millions or billions of nodes and edges. Consequently, optimizing graph applications has become necessary to accommodate the increasingly large quantities of data operated on within such domains. However, the performance of graph programs depends heavily on the input graph structure, underlying algorithm, and utilized hardware, which has caused the optimization of graph programs to be time-consuming and difficult [41, 24].

1.1 Motivation

To address these issues, existing work within the realm of optimizing graph programs targets root issues faced by graph applications, like poor data locality. However, the feasibility and methodology to implement scheduling optimizations that target these issues greatly depend on the backend architecture for which code is being written. For instance, load-balancing techniques on the software-level generally exploit parallelism across machine cores, and can improve cache utilization to reduce the number of slow memory accesses. Despite being software-level, these techniques ultimately manifest themselves as vastly different optimizations within CPU, GPU, and other multicore architectures.

In addition to software-level optimizations, optimizations can leverage features of the specific computer architecture for which code is being written. For a given optimization technique, actual implementation on different architectures can look extremely different. For example, to achieve load balancing, CPU's can enable cache and NUMA partitioning schemes [38, 40], while GPU's can divide active vertices across threads, warps, and CTA's [7]. Furthermore, advances in novel architectures, such as manycore systems and domain-specific accelerators, pose a potential space for new optimizations for graph applications. For instance, Swarm is a domain-specific multicore system that exploits ordered parallelism in graph applications [19].

Several efficient graph analytics frameworks now simplify writing optimized graph applications by leveraging these kinds of compiler and hardware-specific optimizations. Projects like Ligra, Gunrock, and Galois optimize for different situations - Ligra optimizes for shared-memory systems [32], Gunrock optimizes graph applications for GPU's [35], and Galois is highly optimized for road networks [26]. Some frameworks like GraphBLAS [22] and Abelian [12] generate code for both CPU and GPU platforms, but only support a limited set of optimizations or applications. GraphIt, a graph-specific DSL that targets CPU's and GPU's, includes several hardware-independent and dependent optimizations [41, 7, 39], but generalizing it to other architectures remains non-trivial.

Recently, novel manycore architectures - systems composed of hundreds or thousands of cores to leverage thread-level parallelism - have been proposed [2, 8, 13, 34, 29]. These systems include the HammerBlade manycore [8], which leverages software-managed scratchpad memory on its hundreds of small cores to enable parallelism. The flexible core and memory structure expose great optimization potential for graph applications. In addition, many domain-specific accelerators (DSA's) [19, 35, 3, 21] also implement their own specialized compilers to generate code, but this process can be repetitive and complex. To reach the full potential of graph application performance on a variety of graph inputs, a framework must be able to generate code for a diverse set of architectures and easily extend itself to accommodate hardware-specific optimizations.

The Swarm architecture [19, 20] is an example of an architecture that targets

specific applications - it exploits ordered irregular parallelism (parallel workload that is constrained to an order), a feature of many graph applications. However, writing applications for Swarm can be challenging without having extensive knowledge on the patterns of parallelism in the target algorithm and the optimal way to leverage different features exposed in the Swarm model, as shown in hand-tuned implementations of BFS and SSSP in prior work [18]. Extending an existing framework (like GraphIt) to support Swarm could greatly simplify code generation without having to implement a completely new compiler, but doing so would require an existing hardware-agnostic compiler or representation to lower into Swarm-specific transformations and optimizations.

1.2   Contribution

In this thesis, we develop the Universal Graph Framework (UGF) to generate high-performance graph application code for multiple hardware architectures. UGF separates hardware-independent from hardware-dependent components of the compiler to reuse transformations applicable to any backend hardware and support the simple addition of new backends. We introduce a novel hardware-independent intermediate representation (GraphIR), from which programmers can derive and implement their own architecture-specific GraphVM's (compiler backends) for their hardware's optimizations. Because of this portability, UGF can generate highly optimized code for several different backends, including CPU, GPU, the Swarm multicore architecture, and the HammerBlade manycore architecture.

We also present a new extensible scheduling language that supports scheduling options for hardware-independent optimizations, such as edge traversal and parallelization, and can be extended to support programmer-defined scheduling options for architecture-specific optimizations. Scheduling options can support generating performant code for different applications and input graphs, which may have different performance bottlenecks. For example, the nodes in road network graphs usually have lower degrees than nodes in social network graphs, so specifying a push traversal direction (rather than pull) may improve performance in some applications [4, 6].

Next, we implement the Swarm GraphVM, a backend targeting the Swarm mul-

ticore architecture, to demonstrate the feasibility of the UGF design. We generate Swarm C++ code that utilizes Swarm primitives and data structures to speculatively execute tasks using programmer-defined timestamps. To generate correct code from the hardware-independent GraphIR, the Swarm GraphVM performs a series of compiler transformations, including the lowering of shared variable accesses to private accesses and the conversion of frontier-level iterations to vertex-level tasks.

Building off the UGF scheduling language, the Swarm GraphVM also introduces several Swarm-specific optimization parameters, including Swarm task coarsening and Swarm data structure configuration. We motivate these optimizations with results from manually tuned low-level Swarm code, where we explore the optimization space for the Swarm architecture. These optimizations are benchmarked against un-optimized Swarm code generated by the GraphVM to demonstrate UGF's ability to produce not only correct code, but also highly optimized code for different architectures.

1.3 Thesis Organization

In Chapter 2, we discuss the GraphIt DSL compiler, from which UGF derives much of its compiler frontend, and its existing infrastructure for scheduling optimizations and compiler transformations for CPU and GPU architectures. We also discuss Swarm, the target architecture for the Swarm GraphVM.

In Chapter 3, we present the main contributions of UGF, including the GraphIR and novel extensible scheduling language. In Chapter 4, we detail the Swarm GraphVM structure, including several compiler passes required for correct code generation, and how the GraphVM builds off the UGF design. In Chapter 5, we describe efforts to manually optimize Swarm code using low-level Swarm primitives, and how we integrate these findings into an optimization pass in the Swarm GraphVM.

In Chapter 6, we evaluate the performance of code generated by the Swarm GraphVM and analyze the performance of individual optimizations described in Chapter 5. In Chapter 7, we summarize our contributions and provide possible future directions.

Chapter 2

Background

2.1 GraphIt DSL Compiler

The main contribution of this work is the Universal Graph Framework (UGF), built on the existing GraphIt DSL (domain-specific language) compiler for graph analytics. GraphIt is a high-performance graph-specific compiler that enables programmers to convert graph algorithm code into highly optimized architecture-specific code. The GraphIt DSL decouples algorithm code from scheduling optimizations, such as edge traversal direction, vertex deduplication, load-balancing options, and data layout schemes. This separation enables programmers to write optimized graph applications quickly and easily without having to manually explore the entire optimization space for the best implementation.

The compiler consists of three components: the frontend, which parses high-level graph abstractions, graph operators, and user-configured scheduling optimizations; the midend, which consists of multiple lowering passes that use the schedule to transform the program and assert correctness; and the backend, which generates architecture-specific output code. Currently, the GraphIt DSL compiler can generate code for CPU and GPU architectures.

Currently, while the midend includes passes that are necessary for both CPU's and GPU's, it also contains hardware-specific transformations. With the increasing number of multicore and manycore architectures that target graph applications [19, 2, 33, 28],

there has been interest in extending the GraphIt compiler to support more architectures. However, each new architecture introduces novel architecture-specific complexities and optimizations, which GraphIt must also support. UGF seeks to bridge this gap by centralizing the hardware-independent parts of the GraphIt compiler to help programmers easily build highly optimized compiler backends for their own hardware.

2.1.1   Algorithm Language

The first part of the GraphDSL is the algorithm language, which is the set of operators and object abstractions that programmers can use to write GraphIt code. These abstractions are specific to graph applications - for instance, the language uses `VertexSet` and `EdgeSet` to define sets of vertices and edges to process, and `applyModified` and `apply` operators to apply user-defined functions across these sets. Because it is part of the frontend, the algorithm language is hardware-independent, so programmers can use the same algorithm language to generate code for multiple hardware backends. UGF utilizes the same algorithm language as the original GraphIt compiler.

Figure 2-1 shows an example of the algorithm language for BFS. Of note, the edges in the `frontier` edgeset are each updated with the `updateEdge` function using the `applyModified` operator on Line 19. Two schedules, `s0` and `s1` are attached to the `WhileLoopStmt` on Line 18 and the `EdgeSetIterator` on Line 19, respectively.

2.1.2   Scheduling Language

One of the key features of the GraphIt DSL is the separation of the algorithm language from the scheduling language, which allows programmers to fine tune optimization parameters, such as vertex deduplication in vertexsets or edge traversal direction, for their use case. Different input graphs, applications, and target hardware may require vastly different scheduling parameters, so having a separate scheduling language greatly simplifies the optimization process for programmers. For instance, road graphs typically consist of nodes with low in/outdegrees. Consequently, running iterative applications, such as BFS or SSSP, on road graphs feature small frontiers to traverse each round. For applications with small frontiers, it is more optimal to use a push traversal direction, as opposed to a pull direction [4, 6].

```
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 func updateEdge(src : Vertex, dst : Vertex)
11        parent[dst] = src;
12 end
13 func main()
14    var frontier : vertexset{Vertex} = new vertexset{Vertex}(0);
15    var start_vertex : int = atoi(argv[2]);
16    frontier.addVertex(start_vertex);
17    parent[start_vertex] = start_vertex;
18    #s0# while(frontier.getVertexSetSize() != 0)
19       #s1# var output : vertexset{Vertex} =
20          edges.from(frontier).to(toFilter).
21             applyModified(updateEdge, parent, true);
22       delete frontier;
23       frontier = output;
24    end
25    delete frontier;
26 end
```

Figure 2-1: Algorithm specification for breadth-first search (BFS) in GraphIt.

Programmers can also define architecture-specific scheduling optimizations to use in the scheduling language. For instance, GPU architectures perform best on applications with loops that experience large amounts of parallelism on each iteration, but falter on cases where the work of each iteration is outweighed by the cost of launching a GPU kernel for each iteration [7, 35, 31]. As a result, the GraphDSL scheduling language includes a GPU-specific scheduling function for fusing these iterations together to be launched from a single GPU kernel, improving work-efficiency. In the example with road graphs, small frontiers limit the amount of parallelism and work one can achieve in each round, so GPU programmers would opt to enable kernel fusion.

Figure 3-5 shows an example of the Swarm scheduling language used to optimize BFS.

2.1.3 Midend and Code Generation

The existing GraphIt compiler lowers the input program into an AST (abstract syntax tree) representation, and the compiler midend performs a series of analyses

and transformations to optimize the intermediate representation. Currently, it consists of both hardware-independent passes, such as edge traversal direction and data layout transformations, and hardware-dependent passes, like load balancing schemes on GPU's or NUMA on CPU's. To fully optimize for any backend architecture, UGF includes both sets of midend passes, as to fully leverage hardware-independent optimizations while enabling programmers to write hardware-specific optimizations. To illustrate the distinction between the two sets of midend passes, we briefly discuss examples of hardware-independent and dependent optimizations.

Hardware-Independent Passes

Graph applications have a huge optimization space, but implementing every possible combination of optimizations for every application and input graph can be time-consuming. Several of these optimizations can be used irrespective of the underlying hardware. For instance, the hardware-independent portion of the GraphIt midend implements compiler passes for configuring edge traversal direction (push or pull), specifying the layout of vertexset data in memory (bitvector or boolmap array representations), and deduplicating vertices in vertexset frontiers. While the effect of these compiler passes may differ greatly among different hardware backends, they can be utilized by all hardware.

Hardware-Dependent Passes

The GraphIt midend also contains several hardware-dependent passes for CPU and GPU architectures. These passes are specific to just one architecture - every target architecture has a distinct execution model, programming model, and memory layout, so each will also have a distinct set of transformations necessary for optimal and correct code generation. For instance, CPU architectures consist of larger memories, including a large LLC, so they can afford to spread workload across multiple cores and create thread-local priority queues for priority-based graph applications [39, 5, 11]. On the other hand, GPU cores do not have as large of memories, but have significantly more compute power and memory bandwidth [15, 7]. As a result, optimizing edge blocking for data locality and improving data parallelism is crucial to performance on

GPU's [14, 27]. Additionally, the midend includes a GPU-specific pass for identifying reusable vertexset memory, as to reduce unnecessary memory allocations.

These hardware differences lead to different optimizations in the GraphIt midend. UGF reuses these passes, and ultimately ports these hardware-specific optimizations to respective GraphVM's (compiler backends specific to a target architecture).

2.2 Swarm

Parallel graph processing frameworks are often limited by the parallelism that the underlying hardware can feasibly expose. To improve this bottleneck, novel graph-specific accelerators explore novel techniques for better load balancing and work-efficiency from added task parallelism. New manycore architectures add hundreds and thousands of cores to produce additional parallel compute power [2, 8, 29]. Even within these architectures, there is a diverse set of models. Some follow a SPMD (Single Program, Multiple Data) model, where all cores in a group execute the same program (e.g. HammerBlade), but others may follow different models (e.g. SIMT, Single Instruction, Multiple Threads). Each parallel framework also implements its own memory structure, programming model, and execution model. In this work, we focus on the Swarm architecture as one of the new hardware backends that UGF targets, as it introduces yet another layer of parallelism by utilizing speculative execution of tasks.

2.2.1 Architecture

Swarm [19, 20] is a novel multicore architecture that leverages ordered parallelism in programs to speculatively execute fine-grained tasks with assigned timestamps. It uses a task queue unit implemented in hardware to keep track of tasks to execute and a commit queue to hold the speculative state of tasks that are yet to commit. Ideally, tasks are fine-grained enough such that tasks can speculatively spawn multiple rounds into the future and distribute across hundreds of simple cores, thus increasing parallelism more than on conventional CPU's.

Swarm's programming model includes the concept of timestamps, which can be configured by programmers. Users may configure the same timestamp for multiple

tasks, which may run in any order, or they can configure ordered timestamps for tasks which must run in some particular order. For example, tasks processing multiple vertices in a frontier may run with the same programmer-defined timestamp, but tasks across frontier iterations will run with different programmer timestamps. Finally, Swarm converts these timestamps into global timestamps, where it assigns each task a unique timestamp and conveys these order constraints to its hardware. Tasks then run atomically and in timestamp order, and in the case where speculation is incorrect or memory accesses produce an order violation, tasks abort and may re-execute.

2.2.2 Optimizations

To fully take advantage of the parallelism Swarm offers, tasks should be as small and simple as possible to reduce the cost of aborts and to improve load balancing across hundreds of cores. Moreover, small tasks can be optimally distributed to certain cores according to the limited memory they read and write from, improving cache utilization and performance on individual cores. To do this, Swarm can assign spatial hints to tasks to guide task distribution across cores [18]. Hints are often an address of memory that are read or written by tasks - tasks that access the same memory should be assigned to the same cores, as to maximize cache efficiency.

Swarm's model fits well with priority-based algorithms, where order constraints between iteration rounds can prevent other architectures from utilizing all possible parallelism. Non-speculative architectures may explore parallelism within rounds but fail to parallelize across rounds. Because Swarm can speculatively queue vertices multiple rounds into the future, we can now parallelize across rounds, as vertices from multiple frontiers can be processed concurrently while appearing to commit in order.

2.2.3 T4 Compiler

Swarm currently offers a manual low-level Swarm API, where programmers can configure individual tasks, spatial hints, and low-level timestamp options. Additionally, however, the Swarm T4 compiler [37] builds on this low-level API to provide a more high-level method for writing Swarm code. The T4 compiler compiles C++ application code with Swarm extensions - it automatically converts loops to Swarm tasks, attaches

```
 1  swarm::PrioQueue swarm_frontier;
 2  swarm_frontier.for_each_prio([&](unsigned level, int src, auto push) {
 3      src.visited = true;
 4      process_vertex(src);
 5      for (int dst : src.neighbors()) {
 6          if (!dst.visited) {
 7              push(level + 1, dst);
 8          }
 9      }
10  }
```

Figure 2-2: Example of Swarm C++ code for the Swarm T4 Compiler. Vertices are dequeued from the `PrioQueue`, processed on Line 4, and unvisited neighbors are pushed to the `PrioQueue` with an incremented timestamp on Line 7.

spatial hints to qualifying tasks, and assigns timestamps to tasks. UGF targets this version of Swarm C++ for code generation in the Swarm compiler backend.

An example of the Swarm C++ extensions is described in Figure 2-2, where we present an example of how this API may be used to represent a graph application. Here, the "`swarm_frontier`" is a Swarm priority queue that stores tasks to be executed - in this case, the queue stores vertices to process. The "`for_each_prio`" lambda body stores operations to be executed on each vertex, with the target vertex of each task indicated by the "`int src`" parameter. The `push` function on Line 7 enqueues vertices with a programmer-assigned timestamp of "`level + 1`", which, in this case, assigns the next round of vertices to the next round of tasks. Multiple vertices can (and often, will) be assigned the same programmer-assigned timestamp, but the T4 compiler ultimately converts this to a unique virtual timestamp to simplify inter-task conflict detection [19].

2.2.4 Comparison Against CPU

Swarm's ability to speculate across rounds gives it a huge advantage in priority-based algorithms when frontiers tend to be small. If frontiers are small, as in the case of road graphs, parallelism within rounds can be limited. When compared to a normal CPU architecture that cannot speculate, optimized Swarm code achieves nearly 3x speedup on road graphs (Figure 2-3). On the other hand, CPU's tend to be faster in the case of power-law degree distribution graphs. From this comparison, we can conclude that a graph framework should be able to generate code for both architectures, since both

	RN	RC	RU	PK	HW	LJ
BFS	2.59	2.56	2.39	0.38	0.15	0.30
SSSP w/ Delta Stepping	1.57	2.04	1.90	0.38	0.20	0.38

Speedup over Optimized CPU code

Figure 2-3: Performance comparison of Swarm vs. CPU optimized code on BFS and SSSP Δ-Stepping. Green indicates Swarm code speedups over CPU code, while red indicates slowdown. Columns correspond to abbreviated graph inputs (full names listed in Table 6.1), and rows correspond to algorithms.

clearly have strengths in certain situations. This example motivates the design of UGF, which enables generating code for novel architectures that each bring unique optimizations, all from the same input algorithm code. In this work, it specifically motivates the development of the Swarm GraphVM, as to explore the optimization potential of fine-grained, speculative task scheduling with Swarm.

Chapter 3

UGF Design and Implementation

UGF aims to make the existing GraphIt DSL compiler more extensible to new backend architectures while retaining much of its existing hardware-independent optimization space. To make UGF portable, as to support not only a few, but a diverse set of backend architectures, UGF must first separate the hardware-independent from hardware-dependent aspects of the compiler. We accomplish this through the design of the GraphIR representation, a new scheduling language and API, and a novel metadata API that builds on GraphIR to support optimizations or hardware-dependent attributes. The design of UGF is summarized in Figure 3-1.

3.1 GraphIR Representation

To simplify the process of adding new backends to UGF, we design a novel intermediate representation (GraphIR) that is a hardware-independent representation of the input algorithm code. This allows the compiler to support numerous GraphVM's (hardware-dependent compiler backends) that utilize the same GraphIR. GraphIR is now defined in the hardware-independent compiler midend, and is consumed in the hardware-specific GraphVM's.

The GraphIR is an AST representation of the program used in the compiler midend, denoting components of the program as nodes. For example, there are generic nodes such as `WhileLoopStmt` and `VarDecl` (representing while loops and variable declarations, respectively) and graph-specific nodes, such as `EdgeSetIterator` (an
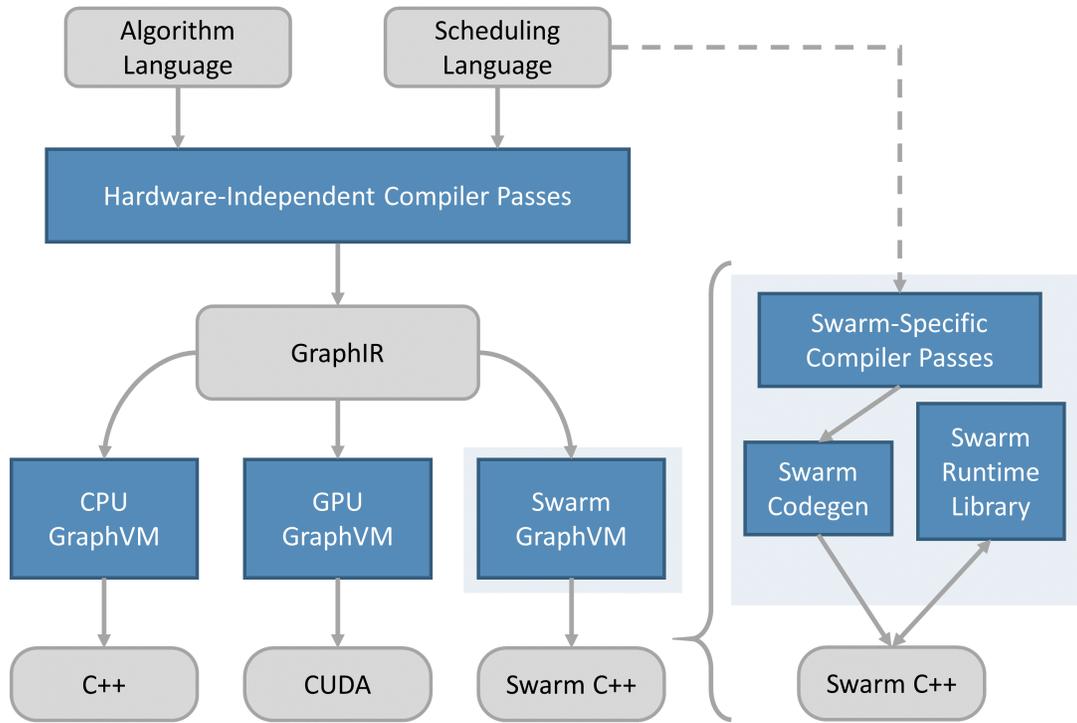
Figure 3-1: An overarching view of the UGF compiler, which includes an algorithm and scheduling specification, hardware-independent transformations and GraphIR, and architecture-specific GraphVM's and output code. An example of the hardware-specific pipeline is included for Swarm, where the scheduling language also contributes Swarm-specific programmer-configured optimizations to the Swarm GraphVM. The GraphVM also includes Swarm-specific code generation and runtime library to support producing Swarm C++.

```
 1  Function updateEdge (int32_t src, int32_t dst,
 2      VertexSet output_frontier, {
 3      bool enqueue = CompareAndSwap<is_atomic=true>(parent[dst], -1, src),
 4      If (enqueue, {
 5          EnqueueVertex(output_frontier, dst)
 6      }, {})
 7  })
 8  Function main (int32_t argc, char* argv[], {
 9      ...
10      WhileLoopStmt<swarm_frontier_convert=true>(VertexSetSize(frontier), {
11          EdgeSetIterator<requires_output=true,
12              frontier_reusable=true,
13              direction=PUSH,
14              is_edge_parallel=true>(
15              edges, frontier, output, updateEdge, toFilter),
16          AssignStmt(frontier, output)
17      }),
18  })
```

Figure 3-2: Optimized GraphIR generated by the compiler for the BFS algorithm with Swarm-specific metadata (e.g. `swarm_frontier_convert` for `WhileLoopStmt`). This text representation is generated by pretty printing the GraphIR, which is an in-memory data structure. Note that the Swarm GraphVM ignores the atomic `CompareAndSwap` specification on Line 3 because Swarm operations always commit atomically.

expression operating on sets of edges in the graph) or `EnqueueVertex` (an expression for enqueuing vertices into a `VertexSet`). We select sufficiently specific abstractions to represent graph applications, but high-level enough to support representation on any hardware. For instance, the two main operators in the GraphIR - `EdgeSetIterator` and `VertexSetIterator` both represent graph-specific loop instructions, and will manifest differently for different architectures in individual GraphVM backends. CPU and GPU backends may explicitly implement edge blocking for the `EdgeSetIterator` instruction to optimize for cache utilization, but the Swarm architecture may opt to simply use a built-in Swarm priority queue structure to iterate over edges.

Initially, these nodes contained both hardware-independent and dependent information. To make the GraphIR truly hardware-independent, we identify each node's "arguments" as parameters independent of architecture and required for correctness. We then leave scheduling options and hardware-specific parameters in the node's "metadata", a flexible container for custom specification described in Section 3.3. An example of this distinction is shown in Figure 3-2, which depicts an example of the GraphIR and usage of the mentioned GraphIR nodes for BFS. Here, the arguments for

`WhileLoopStmt` GraphIR Instruction

Arguments	Metadata used in Swarm GraphVM
bool condition **StmtBlock** body	**bool** swarm_frontier_convert: if the while loop can be converted to use a Swarm queue. **Variable** swarm_frontier_var: holds type and name of Swarm queue variable if while loop can be converted. **StmtBlock** vertex_level_stmts: block of statements that represent vertex-level operations. **StmtBlock** frontier_level_stmts: block of statements that represent frontier-level operations. **List\<Variable\>** global_vars: global variables to pass as parameters into the Swarm queue lambda.

`EdgeSetIterator` GraphIR Instruction

Arguments	Metadata used in Swarm GraphVM
EdgeSet input_graph **VertexSet** input_vset **VertexSet** output_vset **Function** apply_function	**bool** apply_deduplication: whether to deduplicate vertices in output vertexset. **bool** can_reuse_frontier: whether frontier variable and memory can be reused (and can be converted to Swarm queue). **Expr** swarm_coarsening_expr: the numeric expression for the loop coarsening factor if coarsening is enabled. **Expr** spatial_hint: the tensor expression used for the spatial hint for the edge-level applied function if hints are enabled.

Table 3.1: Description of **WhileLoopStmt** and **EdgeSetIterator** GraphIR nodes, separating arguments (left) and hardware-independent scheduling metadata and Swarm-specific metadata (right). A full list of GraphIR nodes and their arguments is located in Appendix B.

the `WhileLoopStmt` contain the condition (checking vertexset size of `frontier`) and the loop's body (Line 10-17), two parameters that are derived from the input algorithm code and must exist regardless of target hardware. `EdgeSetIterator`, likewise, takes in required arguments like the edgeset to be iterated on and the applied function (Lines 11-15).

Metadata, on the other hand, includes several Swarm-specific fields, such as indications of if and how to convert loops to Swarm priority queues (e.g. `swarm_-frontier_convert` in Line 10 of Figure 3-2). Table 3.1 details the breakdown between arguments and metadata for two operators in this example, the `WhileLoopStmt` and the `EdgeSetIterator`. Metadata for the `EdgeSetIterator` also includes optional hardware-independent scheduling parameters, such as frontier deduplication (the option to deduplicate vertices in frontier `VertexSets`) and low-level Swarm optimization parameters.

3.2 Scheduling Language

We also design a new extensible scheduling language for UGF that separates hardware-independent scheduling features, such as edge traversal direction or parallelization, from dependent features, such as NUMA configuration (CPU) or kernel fusion (GPU). We also present a scheduling language API that enables hardware-independent midend passes to utilize hardware-independent scheduling options, but also enables passes in each GraphVM to use hardware-specific scheduling parameters. This contribution enables the creation of the GraphIR, and simplifies the implementation of new GraphVM's.

3.2.1 Scheduling Language Interface

The scheduling language is built using object-oriented design, presenting a hierarchy of `ScheduleObject` classes. The base class (`SimpleScheduleObject`) implements the hardware-independent schedule, consisting of scheduling parameters that are later used for hardware-independent optimizations. From this base class, we extend several hardware-specific schedule object classes that include scheduling parameters relevant to each individual architecture. For example, `SimpleCPUScheduleObject` contains scheduling options specific to the CPU architecture, and `SimpleGPUScheduleObject` contains options specific to the GPU architecture (as shown in Figure 3-3).

The interface also supports the creation and extension of composite schedules, which are schedules that are dependent on some input criteria (e.g. input graph size). These composite schedule objects consist of multiple normal schedule objects, which can then be specific to any hardware backend. This design enables portability of UGF for multiple backends, where each backend can now configure its own scheduling optimizations and inherit from the given hardware-independent schedule objects.

3.2.2 Scheduling Language API

To enable the retrieval of these parameters from the scheduling language objects, we create a novel API that can be used by both hardware-independent and dependent compiler passes. The base `ScheduleObject` class contains hardware-independent configuration and getter functions, such as **configDelta** and **getDelta**, which can be called by
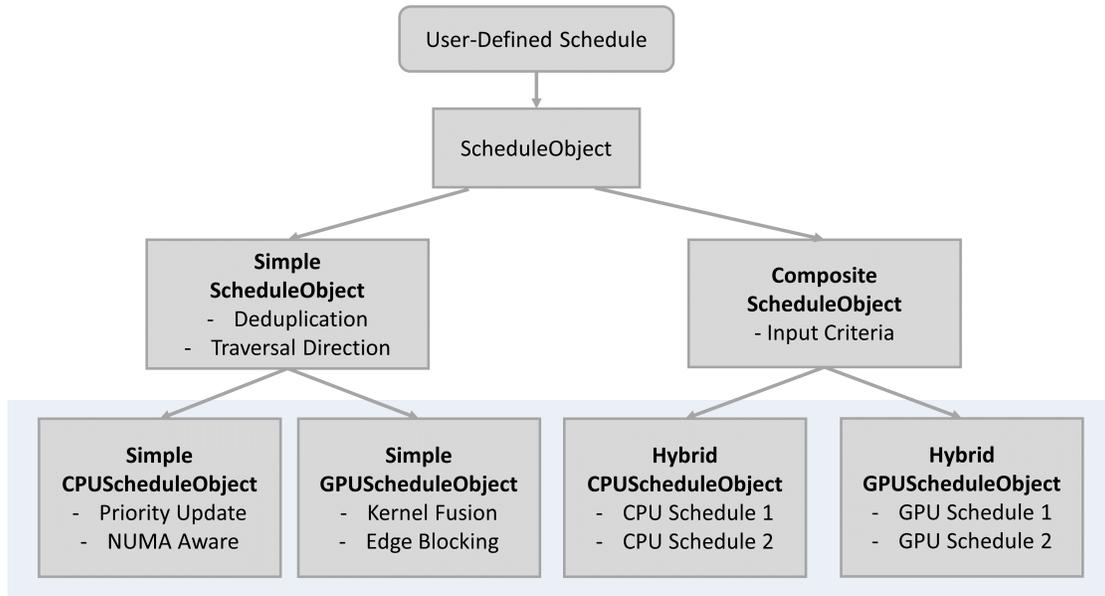
Figure 3-3: Hierarchy of hardware-independent base classes and hardware-specific scheduling classes. Hardware-specific classes are highlighted.
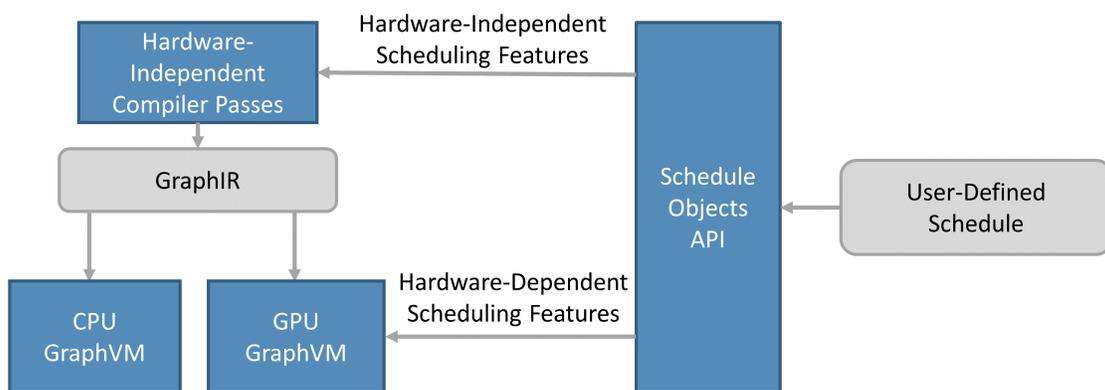
Figure 3-4: Example of Scheduling Language API, interacting with hardware-independent and hardware-dependent aspects of UGF.

Hardware-Independent `SimpleScheduleObject` Options

Function	Options	Description
`configLoadBalance`	`VERTEX_BASED`, `EDGE_BASED`	Configure the load balance scheme to be based on either edges or vertices.
`configDirection`	`PUSH`, `PULL`	Specify either a push or pull traversal for updating vertex data.
`config-Deduplication`	`ENABLED`, `DISABLED`	Enable or disable deduplication of vertices in intermediate or output frontiers.
`configDelta`	`int32_t` or `string` arg	For priority-based algorithms, configure a delta value for priority buckets.

Swarm-Specific `SimpleSwarmScheduleObject` Options

Function	Options	Description
`configQueueType`	`UNORDEREDQUEUE`, `PRIOQUEUE`, `BUCKETQUEUE`	Configure the queue type to represent vertex set iteration in Swarm.
`configLoop-Coarsening`	`ENABLED`, `DISABLED`	Enable or disable loop coarsening for Swarm loop unrolling, using the `SCC_OPT_LOOP_COARSEN_-FACTOR` annotation.
`configSpatialHint`	`ENABLED`, `DISABLED`	Enable or disable the use of spatial hints in a loop iterating on sets of edges or vertices by using the `SCC_OPT_TASK` and `SCC_OPT_CACHELINEHINT` annotations.

Table 3.2: Description of `SimpleScheduleObject` and `SimpleSwarmScheduleObject` functions and options.

hardware-independent passes. Every derived schedule class (`SimpleCPUScheduleObject`, `SimpleGPUScheduleObject` ...) must implement these base functions, which could involve target-agnostic scheduling options to target-specific options. For instance, `SimpleCPUScheduleObject` must support the hardware-independent representation of traversal direction (`PUSH`, `PULL`), so it internally converts its CPU-specific traversal direction options (`SPARSE_PUSH`, `DENSE_PUSH`, `SPARSE_PULL`) to `PUSH` and `PULL` accordingly.

Programmers can then create `config*` functions specific to their target architecture that configure architecture-specific scheduling options, and can create matching getter functions to use in their GraphVM backend. Table 3.2 depicts the delineation between base scheduling functions and Swarm-specific scheduling functions.

```
1  schedule:
2    SimpleSwarmSchedule sched1;
3    sched1.configDeduplication(DISABLED);
4    sched1.configLoopCoarsening(DISABLED);
5    sched1.configSpatialHint(ENABLED);
6    program->applySwarmSchedule("s0:s1", sched1);

7    SimpleSwarmSchedule sched0;
8    sched0->configQueueType(PRIOQUEUE);
9    program->applySwarmSchedule("s0", sched0);
```

(a) Optimal Swarm schedule for BFS on high-diameter graphs, such as road networks.

```
1  schedule:
2    SimpleSwarmSchedule sched1;
3    sched1.configDeduplication(DISABLED);
4    sched1.configLoopCoarsening(ENABLED);
5    sched1.configSpatialHint(ENABLED);
6    program->applySwarmSchedule("s0:s1", sched1);

7    SimpleSwarmSchedule sched0;
8    sched0->configQueueType(PRIOQUEUE);
9    program->applySwarmSchedule("s0", sched0);
```

(b) Optimal Swarm schedule for BFS on power-law degree distribution graphs, such as social graphs.

Figure 3-5: Examples of optimized Swarm schedules for BFS.

3.2.3 Scheduling for Swarm

The Swarm GraphVM supports the use of multiple Swarm API queues, including a built-in priority queue (`PrioQueue`), its extension (`BucketQueue`), and an unordered queue (`UnorderedQueue`), to store and process vertices. In priority-based applications, such as BFS, using either `PrioQueue` or `BucketQueue` will likely yield better performance. However, other more complex applications may incur extraneous overhead from these queues, and will perform better when using a simple unordered queue to represent sets of vertices. This scheduling option allows the user to try multiple queue options and easily accommodate new queue types from the Swarm API.

The scheduling language also supports additional Swarm optimizations that leverage its unique execution model, including the ability to control how workload is divided into tasks and how those tasks are distributed to Swarm cores. Figure 3-5 demonstrates how the `SimpleSwarmSchedule` can leverage hardware-independent scheduling parameters (like deduplication) and Swarm-specific options to optimize BFS. These optimizations are described in-depth in Chapter 5.

3.3 Metadata API

To store optional scheduling parameters or hardware-specific parameters in IR nodes, we develop a metadata map. Programmers building a new UGF backend for their architecture can add their own attributes to metadata maps attached to GraphIR nodes in order to support architecture-specific transformations and code generation. This contrasts with node arguments, which are required for correctness and are hardware-

Function	Description
`getMetadata<T>(str field_name)`	Return the metadata field of type T and specified name from the metadata map.
`setMetadata<T>(str field_name, T field)`	Set a metadata field of type T and specified name by storing the field in the metadata map.
`hasMetadata<T>(str field_name)`	Returns true if the metadata map has a field of type T and specified name. Otherwise, returns false.

Table 3.3: Description of the Metadata API and its functions.

independent. As a result, the metadata map prevents programmers from having to modify the underlying GraphIR nodes themselves for hardware-specific additions.

Examples of use are presented in Chapter 4 for the Swarm backend, where various metadata fields are set in the Swarm-specific compiler passes to indicate how certain portions of code should be generated during code generation.

We then present a simple API for programmers to use to interact with metadata in Table 3.3. Notably, the map is implemented as a flexible container that can support fields of various types. Consequently, the three API functions are also implemented as template functions.

3.4   GraphVM Compiler Backend

UGF includes a series of hardware-independent or reusable passes that can be shared across architectures with commonalities (e.g. Swarm and GPU's, where both can enable the identification of reusable frontiers), derived from the original GraphIt compiler. As seen in Figure 3-1, hardware-specific compiler components are now located in individual GraphVM's, or compiler backends. The GraphVM consumes the hardware-independent GraphIR from Section 3.1 to apply hardware-specific scheduling optimizations and, ultimately, generate architecture-specific code. To support simple development of GraphVM's, UGF includes the metadata API to retrieve parameters from GraphIR nodes and the scheduling API to read the attached scheduling options.

From this framework, programmers can define target-specific compiler passes and code generation to utilize metadata, as mentioned previously, by storing and retrieving hardware-specific parameters and features from GraphIR nodes. Programmers should also include a runtime library in their GraphVM, where optimized, hardware-specific

versions of various graph utility functions are implemented. Code generation in the GraphVM is generally built to call runtime library functions, as to reduce the complexity of the code generation step.

In all, UGF supports a GraphVM for CPU, GPU, Swarm, and the HammerBlade manycore, demonstrating the flexibility of the framework. In the next section, we describe the implementation of the Swarm GraphVM.

Chapter 4

Swarm GraphVM

To demonstrate the extensibility of the UGF model, we develop a novel backend (the Swarm GraphVM) for the Swarm architecture that leverages newly implemented Swarm-specific compiler transformations. The Swarm GraphVM consumes input GraphIR and generates C++ with Swarm extensions, which can be compiled by the Swarm T4 compiler to produce tasks for the Swarm hardware. In this section, we focus on the set of Swarm-specific transformations, as hardware-independent passes have been discussed in previous sections and works [41].

As mentioned, the Swarm execution model relies on small tasks operating on individual vertices, while minimizing dependencies between tasks to avoid large, costly aborts. As such, the Swarm-specific compiler transformations implemented for the Swarm backend fall into two main categories: the division of vertex frontier iterations into individual vertex-level tasks, and the conversion of shared variables to task-local, private ones. We also present a transformation pass for the deduplication of vertices in frontiers, which is often required for correctness. The flow of compiler passes is detailed in Figure 4-1.

4.1   Frontier Task Division

The frontier pattern is common in graph applications, where algorithms such as BFS or BC process sets of vertices in the current frontier, store their neighbors in the next frontier, and then traverse the next frontier's vertices in the next iteration. An
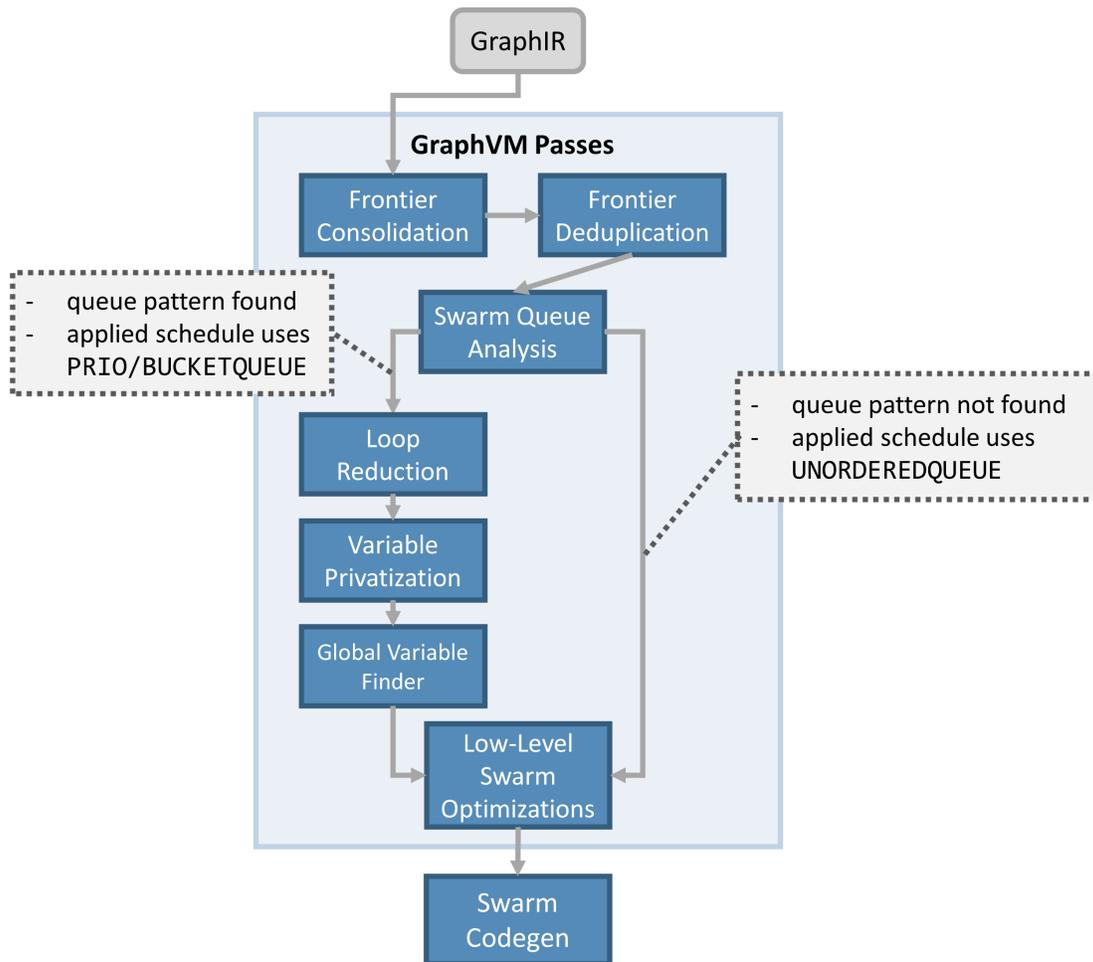
Figure 4-1: Flowchart of compiler transformations and optimizations in the Swarm GraphVM. Frontier Consolidation, Swarm Queue Analysis, and Loop Reduction are described in Section 4.1. Variable Privatization and Global Variable Finder are described in Section 4.2. Frontier Deduplication is described in Section 4.3. Low-Level Swarm Optimizations are described in Section 5.2.

```
1    while (frontier.getVertexSetSize() != 0)
2        var output : vertexset{Vertex} = edges.from(frontier).applyModified(
             updateEdge);
3        delete frontier;
4        frontier = output;
5    end
```

(a) Input GraphIt code for frontier iteration, applying the `updateEdge` function onto each frontier.

```
1    WhileLoopStmt<swarm_frontier_convert=true>(VertexSetSize(frontier), {
2        EdgeSetIterator<frontier_reusable=true>(edges, frontier, output,
             updateEdge),
3        AssignStmt(frontier, output)
4    });
```

(b) GraphIR representation of above GraphIt code.

```
1    WhileLoopStmt<swarm_frontier_convert=true>(VertexSetSize(frontier), {
2        EdgeSetIterator<frontier_reusable=true>(edges, frontier, frontier,
             updateEdge),
3    });
```

(c) Lowered GraphIR following frontier reuse and consolidation.

Figure 4-2: GraphIR example and transformation following frontier reuse and consolidation pass.

example of this is depicted in Figure 4-2. Many architectures can parallelize the processing of vertices within each frontier, but Swarm provides a unique opportunity to also parallelize across frontier rounds, given that tasks can execute speculatively across these rounds. To achieve this speculation, we implement a series of analysis and transformation passes that convert loops that operate on entire frontiers to Swarm timestamp-ordered tasks that operate on a single vertex.

4.1.1 Frontier Consolidation

First, we implement a pass to remove intermediate frontier variables and consolidate frontier usage to remove unnecessary dependences between rounds. Eliminating extraneous frontiers also simplifies the remaining steps to convert frontier-based loops to vertex tasks. UGF supports reusing passes across different backends, so to identify these frontier patterns, we reuse the FrontierReuseAnalysis pass also used in the GPU GraphVM. This pass is a liveness analysis used to check if the memory allocated to a

vertex frontier is reusable between rounds. Having a reusable frontier implies that we can eliminate redundant, intermediate frontiers, leaving just one active frontier within the loop and making it safe to convert to a Swarm queue.

Once a reusable frontier is identified by a metadata flag set in the FrontierReuse-Analysis pass, we can safely transform the program to modify the frontier in-place without an intermediate output variable. For instance, in Figure 4-2a, we find an intermediate `output` frontier, but since `frontier` was marked as reusable when lowered to GraphIR, we consolidate the two frontiers, as in Figure 4-2c. Importantly, this transformation also removes the dependence that the `output` variable would enforce between tasks of different iterations, and thus allows Swarm to efficiently speculate across rounds.

4.1.2 Swarm Queue Analysis

The previous transformation also leads into the next analysis pass, which checks for while loops with the frontier pattern that can be transformed into Swarm priority queues. When a loop iterates and operates only on one frontier, the compiler can safely convert this to a Swarm queue loop that reduces each frontier iteration to a series of per-vertex operations. Once this frontier pattern is detected, the analysis generates code according to the Swarm C++ API from Section 2.2.3 to transform while loops to utilize a Swarm priority queue. We find that the UGF and GraphIR design succeeds at presenting the `WhileLoopStmt` and `EdgeSetIterator` abstractions, which easily map to a Swarm queue loop. Moreover, these abstract nodes can flexibly hold the `frontier_reusable` metadata field to help this Swarm-specific pass identify a convertible loop.

In this pass, the Swarm GraphVM converts the enqueuing of vertices into the spawning of tasks that handle these operations. For correctness, however, these tasks still must be assigned timestamps that enforce order across frontier rounds. Assigned timestamps consequently correspond to the round that the vertex would have been dequeued. In a frontier pattern, tasks will be assigned incremented timestamps, will execute speculatively, but still appear to commit in-order, thus preserving correctness.

```
1    while (frontier.getVertexSetSize() != 0)
2       (0) frontier = edges.from(frontier).
           applyModified(updateEdge);
3       (1) frontier.apply(mark_visited);
4    end
```

(a) Input GraphIt code with two loop body operations.

```
1    swarm::PrioQueue swarm_frontier;
2    swarm_frontier.for_each_prio([](unsigned
         level, int src, auto push) {
3      switch (case % 2) {
4        case 0:
5          for (dst : src.neighbors):
6            updateEdge(src, dst);
7            push(level + 1, dst);
8            break;
9        case 1:
10           mark_visited(src);
11           push(level+1, src);
12           break;
13     }
14   });
```

(b) Generated Swarm code with two switch cases for the two loop body operations.

Figure 4-3: Input GraphIt and output Swarm C++ code for a frontier loop body with multiple operations. `EdgeSetIterator` step 0 in (a) matches the generated switch case for case 0 in (b), and the `VertexSetIterator` step 1 in (a) matches the generated switch case for case 1 in (b).

4.1.3   Frontier Loop Body Reduction

Reduction of Vertex Operations to Individual Tasks

The first two compiler passes convert frontier iterations into vertex level tasks. However, if the loop body specifies several ordered operations to be performed on each vertex, these tasks can be further reduced into smaller tasks. After this transformation, each task will execute a single operation on an individual vertex, as demonstrated in Figure 4-3, where the two operations to be performed on a vertex are divided into two separate tasks.

To achieve this structure, the Swarm GraphVM transforms the loop body into individual switch statements. Each individual step is then assigned an incremented timestamp, as to enforce the order of these operations. Consequently, tasks representing each operation of the loop body will still commit in order, which is crucial for correctness. For instance, in Figure 4-3, the `EdgeSetIterator` operation in step 0 transforms into tasks with level 0. The tasks in switch case 0 then enqueue tasks with level 1 for the `VertexSetIterator` from step 1. These tasks enqueue tasks with level 2, which execute the `EdgeSetIterator` operation on the next frontier, thus preserving order both within the loop body and across loop iterations.

(a)

```
while (frontier.getVertexSetSize() != 0)
    (0) round = round + 1;
    (1) frontier = edges.from(frontier).
            applyModified(updateEdge);
    (2) frontier.apply(markVisited);
    (3) frontier_list.insert(frontier);
end
```

(b)

```
swarm::BucketQueue frontier;
frontier.for_each_prio([&](int i, int src,
auto push) {
    switch (i % 4) {
        case 0:  // empty task
            push(i + 1, src);

        case 1:
            for (edge : src.edges):
                if (!visited(edge.dst)) {
                    updateEdge(src, edge.dst);
                    push(i + 1, edge.dst);

        case 2:
            markVisited(src);
            push(i + 1, src);

        case 3:
            frontier_list.insert(src,
                frontier_list.current_round);
            push(i + 1, src);

}, [&round](...) {
    switch (i % 4) {
        case 0:
            round = round + 1;
            push(i + 1, src);
        ...
        case 3:
            frontier_list.update_size(
                frontier_list.current_round + 1);
            push(i + 1, src);
    }
});
```

(c)

Figure 4-4: (a) Input GraphIt and (b) generated Swarm C++ code for betweenness centrality (BC). (c) Flow of tasks for both per-vertex tasks (left, in blue) and per-frontier tasks (right, in gray), separated by corresponding step in GraphIt code (or corresponding case in generated code).

Identification of Vertex and Frontier Level Tasks

Swarm enqueues tasks for each vertex operation - however, programmers may also want to define operations that execute once per frontier iteration, rather than once per vertex. Examples of such operations include global variable increments that only occur once per round (Figure 4-4a, Step 0). To accommodate this pattern, the Swarm C++ API provides a `BucketQueue` with two lambda bodies that separate per-vertex and per-frontier tasks. Operations defined in the second, per-frontier body execute only after all the tasks in the first, per-vertex body with the matching case number are executed.

In this half of this transformation, the Swarm GraphVM identifies operations within loops that are per-vertex and per-frontier and stores each group in separate metadata fields. These two buckets are stored as metadata in the `WhileLoopStmt`

42
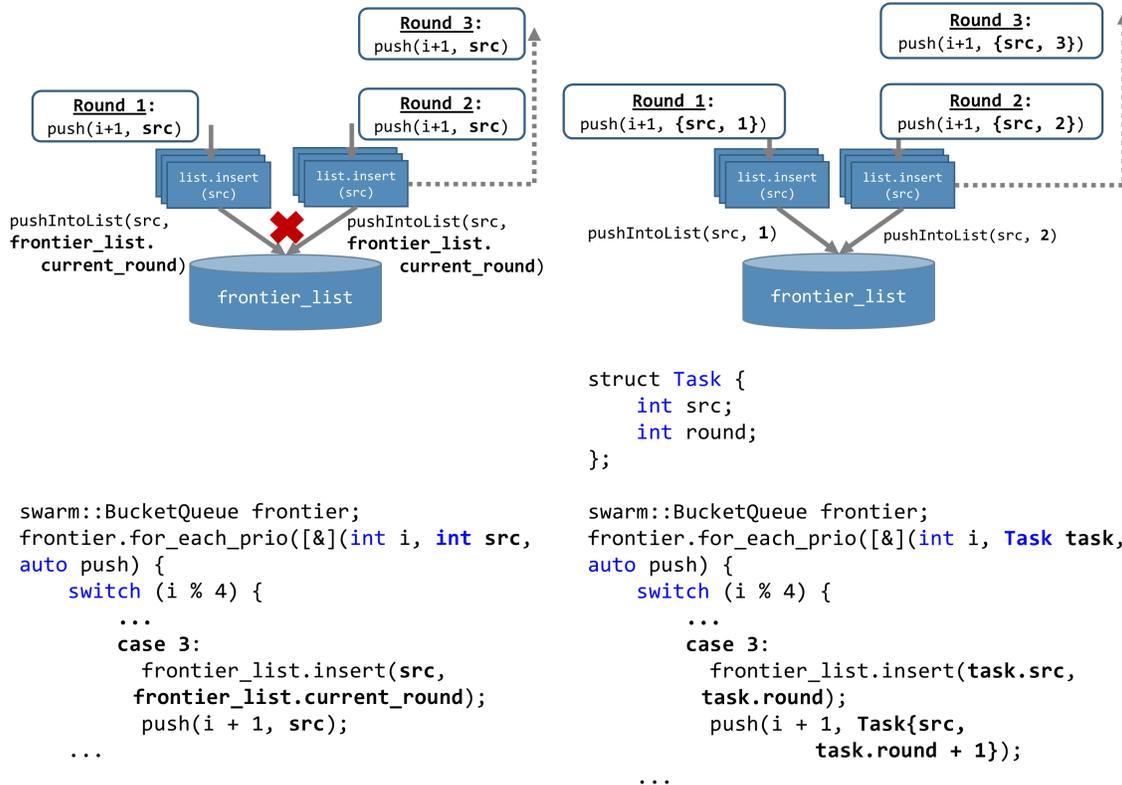
Figure 4-5: Shared to private state conversion when inserting vertices into a frontier list (Step 3 for betweenness centrality (BC) in Figure 4-4a). Index of frontier where vertex should be inserted is converted from shared access (`frontier_list.current_round`) to task-private functional parameter (`task.round`). Corresponding code transformation is shown in bottom half of figure.

node (as shown in Table 3.1), which can be accessed during Swarm code generation to organize statements into the correct lambda bodies (Figure 4-4b).

4.2 Privatization of Shared Variables

When accessing shared variables or state, Swarm must be careful to not introduce unnecessary dependences between tasks. Tasks that operate with private variables will minimize shared reads and writes and generally perform better, as they are more independent from other tasks. However, in graph applications, current loop iterations may depend on previous iterations to run correctly, which will require special handling to prevent unnecessary task ordering and aborts. Additionally, many applications will involve accessing shared data structures to store and update data, introducing new inter-task dependences. For instance, inserting a vertex into a growing list of frontiers requires each task to know which frontier to insert a vertex. The naive implementation inserts vertices into the last frontier in the frontier list, but because tasks run speculatively, tasks from different rounds will compete for the next position in the list to insert a frontier. This conflict creates a false dependence between tasks and prevents speculation across rounds.

The Swarm GraphVM addresses shared inter-task dependencies in two passes. First, the GraphVM lowers inter-task dependences to task-local private state arguments to prevent false data dependences between tasks. Second, the GraphVM analyzes each frontier-iterating loop to identify global variables to pass into the Swarm queue lambda body.

4.2.1 Variable and State Privatization Pass

First, the Swarm GraphVM detects dependences that occur between tasks, such as frontier list size increments, where all tasks insert or modify data within the same data structure. We opt for a functional approach, converting inter-task dependent parameters to function arguments to spawned tasks. This conversion converts previously shared dependences to task-local state variables.

Figure 4-5 presents an example of this pattern in frontier list insertions, where each task depends on previous tasks to determine where to insert a vertex into the data

structure. To prevent this conflict from affecting speculation across rounds, we lower this shared task-to-task dependence to a local, private task variable that stores the frontier list index to insert the vertex into. As seen in Figure 4-5, we demonstrate this conversion in the `Task` argument, `round`. This local incremented variable is passed from task to task as a function parameter, which removes the need for any global accesses. In the general case, the Swarm GraphVM detects these accesses to shared state and easily lowers them to private copies by adding functional parameters passed from task to task.

To insert vertices into a single frontier, tasks from the same round may conflict on the pointer to the next empty space in the frontier. This enforces an unnecessary ordering of tasks that reduces Swarm's ability to parallelize tasks in a single round. The Swarm GraphVM solves this problem by simply using the `UnorderedQueue` structure from the Swarm API inside the Swarm GraphVM runtime library to represent each frontier. This queue allows tasks to append vertices without enforcing an order.

4.2.2   Global Variable Finder Analysis

Additionally, we implement a global variable analysis pass to detect other global variables in frontier-iterating loops. Global variables that do not pose a dependence between tasks, such as the `round` variable on Step 0 in Figure 4-4a, must be specially passed as a parameter into the lambda body of the Swarm queue (Figure 4-4b).

Both shared variables and global variables are detected in this compiler pass and stored as metadata in the corresponding `WhileLoopStmt` node. Code generation reads these metadata fields to build `struct` objects for tasks, and locally modify task-private parameters before generating push statements that functionally pass these parameters to the next tasks.

4.3   Frontier Deduplication

Finally, we address passes that deal with correctness. Here, we develop a novel compiler transformation that deduplicates vertices in frontiers. Deduplication is required in some applications, such as betweenness centrality, to prevent the reprocessing of vertices, but it can also be used in applications such as BFS or SSSP as an optional optimization.

Case 1 in Figure 4-4b depicts an example of deduplication for BC, which requires deduplication for correctness when pushing vertices into the next frontier.

Chapter 5

Swarm Optimizations

Following the Swarm GraphVM transformations presented in the previous section, we now discuss the Low-Level Swarm Optimizations pass (the last pass in Figure 4-1), which reads Swarm scheduling options to enable various low-level optimizations for Swarm and more finely configure the behavior of the generated Swarm C++ code. These options were motivated by hand-tuned implementation experiments which explore potential optimizations that leverage Swarm's architecture (Section 5.1). Because Swarm can benefit greatly from small tasks and optimal load distribution for cache efficiency, we mainly explore (1) optimal task division and assignment of spatial hints to tasks and (2) the optimal granularity to split a loop iterating across vertices or edges. Guided by results from hand-tuned implementation experiments, we then discuss the integration of these optimizations into the GraphVM (Section 5.2).

5.1   Manual Swarm API Optimizations

To explore the optimization space of Swarm applications, we manually implement and tune graph applications written for the Swarm architecture, using low-level Swarm data structures and primitives. While the Swarm GraphVM generates code for the T4 compiler, we found it initially difficult to experiment with Swarm optimizations using T4 compiled code, as its design trades increased simplicity for less customizability in defining task boundaries and hint assignment. Consequently, this section focuses on code written with the low-level Swarm API, which allows us to define fine-grained

47

```
1   void writeTask(int level, int ngh, int src) {
2         if (parent[ngh] == -1) {
3               parent[ngh] = src;
4               swarm::enqueue(readTask, level+1, EnqFlags::SAMEHINT, ngh);
5         }
6   }
7
8   void readTask(int level, int src) {
9         ...
10        swarm::enqueue(writeTask, level, Hint(&(parent[ngh])), ngh, src);
11  }
```

Figure 5-1: Example of manually implemented, hand-tuned Swarm code that utilizes the low-level Swarm API. Two tasks (write operation on top, read operation on bottom) are shown.

tasks, explicitly enqueue children tasks with specific timestamps, and assess the effect of different spatial hint schemes. In the example presented in Figure 5-1, tasks are defined as functions in **writeTask** and **readTask**, children tasks are enqueued with the **swarm::enqueue** function, and spatial hints are assigned with **Hint** or propagated with **EnqFlags::SAMEHINT**. We implement hand-tuned programs using this API that explore each of the target optimizations individually, and performance results are reported in Section 6.4.

5.1.1   Spatial Hint Assignment

We first manually implement programs with spatial hints assigned to every task, as to optimally distribute tasks to maximize task data locality. These manual implementations also explicitly define the boundaries between distinct, fine-grained tasks in the program. After all, dividing tasks into more fine-grained tasks allows programmers to assign spatial hints more strategically to small tasks.

Many graph applications follow a read-write-enqueue pattern: information on a source vertex is read, a resulting value is written to some memory address, and the neighbors are enqueued for the next application iteration. For instance, in BFS, we read each outgoing edge from a source vertex, write the updated distance to each destination vertex, and enqueue updated destination vertices for the next iteration. To optimize for locality in this pattern, the source vertex read section of an iteration should be isolated into one task, as all read tasks for this vertex will read the same memory; the write section follows similarly. Thus, a natural task boundary arises

between the read and write step of each loop iteration. In the manually implemented example from Figure 5-1, we explicitly define the boundaries between these tasks by assigning each of these steps to one task.

With fine-grained tasks, we may now assign spatial hints, which often match a memory address that the task reads from or writes to. Hints for the read task should be related to the source vertex, and hints for the write task should be related to the destination vertex. In Figure 5-1, the write task for vertex `ngh` is assigned a hint of `parent[ngh]`, the location that the task writes to. The hint is then propagated to the next read task, which reads information from vertex `ngh`.

5.1.2 Dynamic Task Partitioning

Next, we manually implement programs with loops that process a specified number of edges to maximize task data locality across cores while accounting for the degree distribution of the input graph. As mentioned, Swarm benefits from fine-grained tasks because of reduced cost of aborts and more efficient task distribution across cores due to spatial hint assignment and cache utilization. At the same time, however, making the assigned work for each task too fine-grained can remove potential benefits from bundling similar tasks together. Tasks that access consecutive memory addresses might benefit from bundling to improve cache utilization because consecutive values will be stored in the same cache line. Without coarsening, these tasks would have been distributed to the same core in the fine-grained model anyways. Directly bundling tasks into one task would remove the overhead of enqueuing several tasks while still benefiting from optimal cache retrieval. To manually implement this behavior, the low-level Swarm API enables us to process a specified number of edges together in one loop, where the number of edges is conditional on the degree of the source vertex.

5.2 Integration into Swarm GraphVM

Guided by the results from these manual optimizations, the Swarm API now exposes several code annotations (`SCC_OPT_*` Swarm annotations mentioned in Table 3.2) that allow the Swarm GraphVM to easily leverage these key features of the hardware, including explicit task boundary annotations to separate tasks, spatial hint annota-

tions to specify hint addresses, and loop coarsening annotations to specify Swarm's granularity when splitting loops into tasks.

We thus present the Low-Level Swarm Optimizations pass in the Swarm GraphVM, which consumes the configured Swarm scheduling options to generate these annotations in the output code. As mentioned in Section 3.2.3, we extend the Swarm scheduling language to support the insertion of these annotations by introducing various configuration functions, which are applicable to `EdgeSetIterator` expressions. One configuration function is defined for each of the described optimizations.

The `EdgeSetIterator` expression is the target GraphIR node for these optimizations because it naturally follows the read-write-enqueue pattern described in the previous section, where source information is read and destination vertices are written to by the applied function. These expressions also loop over sets of edges. Consequently, the hand-tuned implementations from the previous section clearly define where task boundary, spatial hint, and loop coarsening annotations should be inserted into the generated code, if the respective scheduling options are enabled.

First, enabling `SPATIAL_HINT` on an `EdgeSetIterator` expression allows the GraphVM pass to insert Swarm task boundary and spatial hint annotations in between the target function's source read and destination write sections. Spatial hints are assigned by searching for a tensor value modified in the applied function to pass into the generated hint annotation. If no potential value is found, then the spatial hint option simply inserts a task boundary, which automatically decreases the size of tasks in the application code. This boundary matches the boundary between read and write tasks in the manually implemented code from Section 5.1.1.

Second, enabling `LOOP_COARSENING` on an `EdgeSetIterator` expression indicates to the GraphVM to insert a loop coarsening annotation prior to iterating across the edges in the edgeset. By default, the loop coarsening option bundles the maximum number of tasks by calculating the number of values that can reside on one cache line (`SWARM_CACHE_LINE / sizeof(type)`). This annotation enables the Swarm GraphVM to mirror the behavior described in Section 5.1.2, such that groups of edges are now processed together in one task, as opposed to individual tasks.

```
1  frontier.for_each_prio([](int level, int src) {
2      SCC_OPT_LOOP_COARSEN_FACTOR(SWARM_CACHE_LINE/sizeof(int))
3      for (int edgeID : neighbors(src)) {
4              int dst = edgeDst[edgeID];
5              SCC_OPT_TASK();
6              SCC_OPT_CACHELINEHINT(&(parent[dst]));
7              if (parent[dst] == -1) {
8                      parent[dst] = src;
9                      push(level + 1, dst);
10              }
11      }
12  });
```

Figure 5-2: Optimized Swarm code with `SCC_OPT_*` annotations for BFS.

Notably, because the hardware-independent scheduling parameters are now separated from the Swarm-specific parameters in UGF, adding and using new parameters does not affect the rest of the compiler. In the Low-Level Swarm Optimizations pass, we check whether either of these Swarm-specific options were enabled, and we simply store annotation argument expressions in the metadata of the corresponding `EdgeSetIterator` node. Thus, the portable design of UGF from Chapter 3 successfully allows for the integration of Swarm-specific optimizations without modifying the underlying hardware-independent compiler.

Figure 5-2 presents a Swarm GraphVM generated example of a transformed `EdgeSetIterator` loop from BFS, where the edge iterating loop is coarsened and the read and write steps are separated by a task boundary on Line 5. A hint is added on Line 6 according to a value (`parent[dst]`) that is read and modified in the separated task.

Chapter 6

Evaluation

We evaluate the performance of the UGF Swarm GraphVM by demonstrating its ability to produce highly optimized Swarm code. We compare the performance of fully optimized generated Swarm code with baseline, unoptimized generated code on 5 graph algorithms and 10 different graph inputs. We also analyze the effect of individual optimizations on performance, and how different combinations of optimizations may benefit different sets of graphs or applications. Baseline code is generated using the default schedule for the Swarm GraphVM (listed in Appendix A.1). For the optimized code, we tune optimal schedules for each application or graph input (listed in Appendix A.2), but we generate code using the same input algorithm code.

6.1 Methodology

We evaluate the Swarm GraphVM by running generated code on the Swarm architectural simulator [37], utilizing a 64-core Swarm CPU configuration, as in prior work [37, 1], with 32KB/core of L1 cache, 1MB/tile of L2 cache and a shared 64MB L3 cache, each with a 64 byte cache line size.

6.1.1 Datasets

Table 6.1 lists the input graphs used in the evaluation experiments, along with graph sizes and download sources. We evaluate the Swarm GraphVM on a variety of graphs, including social networks, road networks, and web graphs. Out of the 10 graphs, Orkut (OK), Twitter (TW), LiveJournal (LJ), Sinaweibo (SW), Hollywood (HW), Pokec

Graph	Vertex count	Edge count
RN [23]	1,971,281	5,533,214
RC [10]	14,081,816	33,866,826
RU [10]	23,947,347	57,708,624
PK [23]	1,632,803	30,622,564
HW [9]	1,139,905	112,751,422
LJ [23]	4,847,571	85,702,474
OK [30]	2,997,166	212,698,418
IC [9]	7,414,865	301,969,638
TW [30]	21,297,772	530,051,090
SW [30]	58,655,849	522,642,066

Table 6.1: Graph inputs used for evaluation. Each undirected edge is counted twice, once per direction.

(PK), and Indochina (IC) have power-law degree distributions, while RoadNet-CA (RN), RoadCentral (RC), and RoadUSA (RU) are road networks with bounded degree distributions and high diameter.

6.1.2   Algorithms

We evaluate the Swarm GraphVM on five applications: PageRank (PR), Connected Components (CC), Breadth First Search (BFS), SSSP with Delta Stepping (SSSP), and Betweenness Centrality (BC). PR and CC are topology-driven algorithms (all edges are traversed during each iteration). BFS, SSSP, and BC are data-driven algorithms (only a subset of active vertices is visited each iteration). SSSP is also a priority-based algorithm, where vertices are processed and tasks are committed in priority order. Each experiment uses the same algorithm code for the above applications, and each uses the same experiment parameters (start node, delta value).

6.1.3   Schedules

Baseline (default) schedules are defined in Appendix A.1. Optimized schedules for each application and graph combination are also listed in Appendix A.2, and were motivated heavily by findings in Chapter 5 and manual tuning.

	RN	RC	RU	PK	HW	LJ	OK	IC	TW	SW
PR	5.63	1.71	2.78	2.25	2.99	2.63	2.41	1.61	2.49	1.33
CC	8.00	2.71	4.95	1.17	1.71	1.22	2.41	1.24	1.33	1.72
BC	1.14	1.08	1.10	1.05	1.28	1.03	1.03	1.36	1.00	1.00
SSSP	1.06	1.03	1.05	1.07	1.00	1.04	1.06	1.00	1.00	1.00
BFS	1.20	1.15	1.18	1.10	1.06	1.05	1.20	1.06	1.08	1.05

Speedup against Swarm Baseline
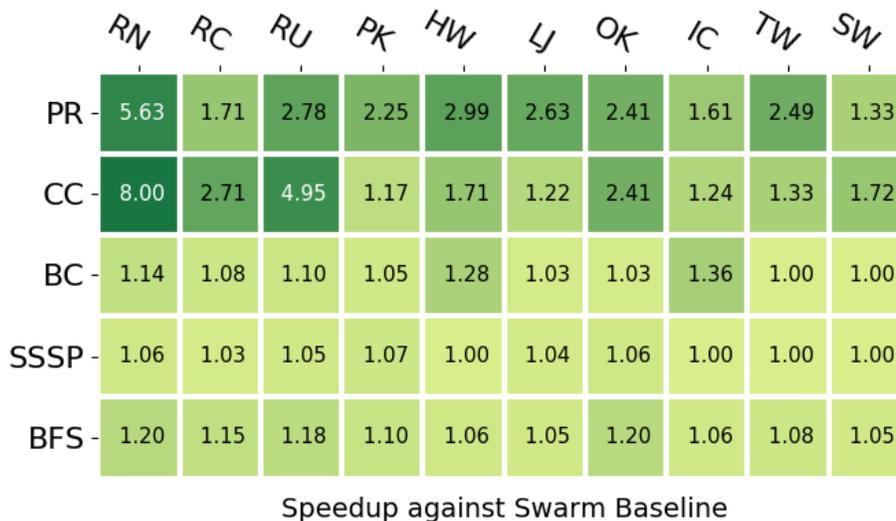
Figure 6-1: Heatmap of speedups of generated code from the Swarm GraphVM. Each cell reports the speedup of the optimized code over the Swarm baseline unoptimized version, with larger speedups in darker green. Columns correspond to graph inputs, and rows correspond to algorithms.

6.2 Performance of Optimized versus Baseline Swarm Code

The Swarm baseline code utilizes default settings without additional task or spatial hint annotations. When compiling this code, however, the Swarm T4 compiler automatically applies several optimizations to generate more work-efficiency and task distribution to this default implementation. Despite this, the Swarm GraphVM is still able to generate code that, when compiled by the T4 compiler, performs significantly faster than the baseline code across all applications, when using an optimally tuned schedule.

In Figure 6-1, we observe consistent speedups in performance, especially in the topology-driven algorithms (PR) and (CC). Because these two applications visit all edges in each iteration, schedules can opt to coarsen each task to process multiple, consecutive edges in one task, which can yield huge cache utilization benefits. For applications that can experience highly contested memory accesses from high in-degree nodes (CC), schedules can also opt to traverse edges in non-consecutive order, as to
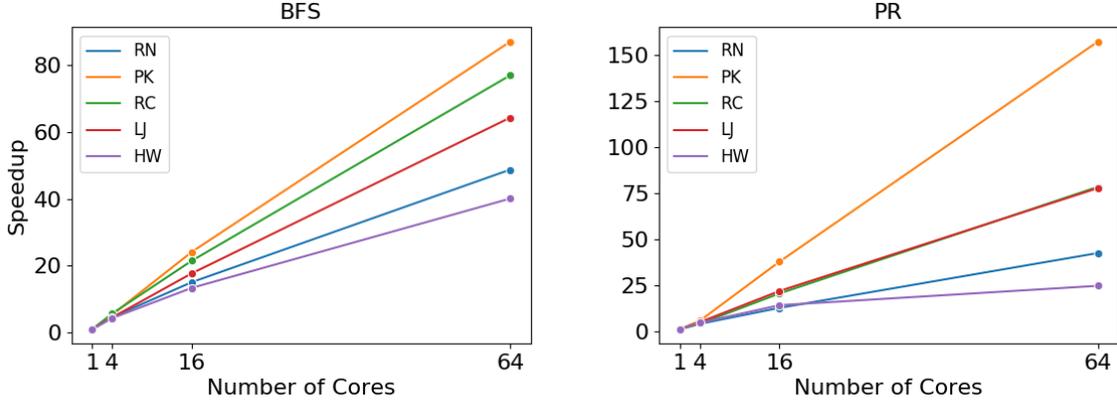
Figure 6-2: Scalability of Swarm GraphVM generated code on 4 core configurations for BFS and PageRank.

trade off worse locality for less competing memory accesses and task aborts.

Data-driven algorithms also exhibit speedups across the map. These applications feature several iterations across frontiers of vertices, where each round must be processed and committed before the next round begins. Swarm achieves speedup in these applications by leveraging fine-grained tasks - tasks can process individual vertices instead of entire sets of vertices from a frontier, which improves work efficiency. Additionally, Swarm removes the data dependence and synchronization requirement across frontier rounds, so tasks can now speculatively execute across rounds. Utilizing spatial hints in these applications to specify cores to execute certain tasks also reduces cache line ping-ponging, thus improving cache utilization on individual cores. The Swarm GraphVM successfully handles these transformations and achieves up to 36% speedup in BFS, BC, and SSSP (as seen in Figure 6-1).

6.3   Scalability and Efficiency of Swarm GraphVM

The Swarm GraphVM not only generates optimized code that is faster than default scheduled Swarm code, but also generates code that utilizes the Swarm architecture well. We evaluate utilization by analyzing the scalability and work-efficiency of the generated code.

First, we evaluate the scalability of code generated by the Swarm GraphVM by running experiments using 1, 4, 16, and 64 core configurations, with appropriately
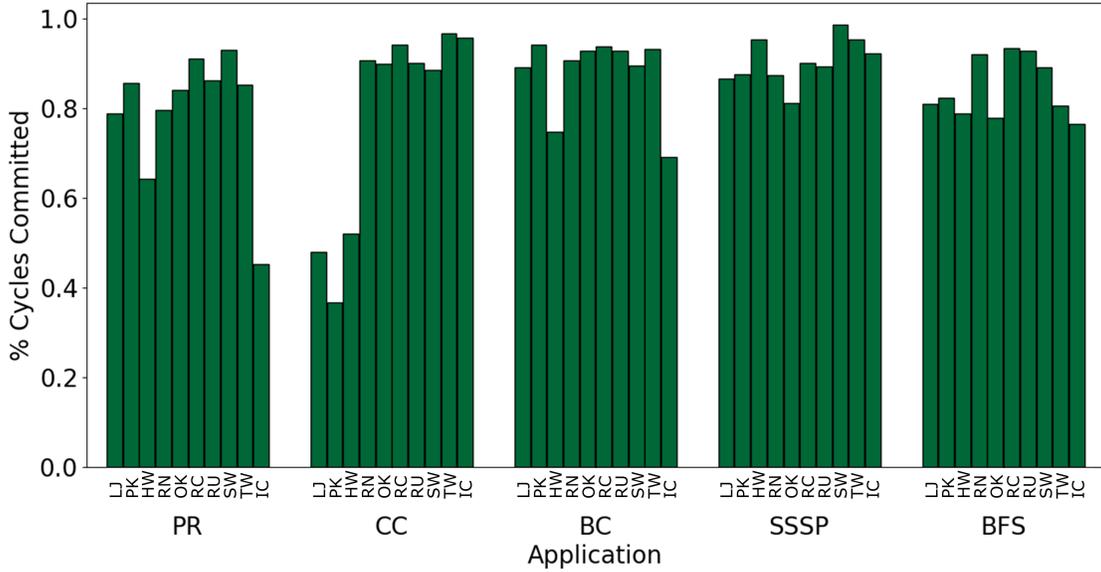
Figure 6-3: Percentage of task execution cycles spent on work that is eventually committed by Swarm cores during execution of generated Swarm code.

scaled cache quantities. In Figure 6-2, we find that code scales linearly with additional cores, which suggests that the GraphVM successfully generates code with the fine-grained parallelism required to fully utilize the multicore structure of the Swarm architecture. Generally, with more cores, multicore architectures may begin to trade off load balancing and synchronization overheads with increased parallelism benefits, but the continuous linear growth in performance strongly supports the scalability of Swarm and the generated code.

We also study the efficiency of individual programs by determining the percentage of task execution cycles that are eventually committed in the five applications, as opposed to cycles spent on work that is eventually aborted. With the optimized schedules, cores consistently spend more than 80% of their time executing cycles that are eventually committed, as shown in Figure 6-3. This suggests that the Swarm GraphVM generates optimized code that saturates all available cores with useful work, across all applications and graph types.

Graph	Application	Manual Code Speedup	GraphVM Code Speedup
RN	BFS	**1.45**	1.17
	SSSP	**1.69**	1.06
PK	BFS	**3.11**	1.09
	SSSP	**3.41**	0.99
LJ	BFS	**2.43**	1.08
	SSSP	**3.30**	1.03

Table 6.2: Speedup in manually implemented and GraphVM generated BFS and SSSP with spatial hints added, when compared to code without any explicitly defined hints. Manual code uses SSSP Bellman-Ford and GraphVM code uses SSSP Δ-Stepping.

Graph	Application	Cycles/Write Task (with hints)	Cycles/Write Task (without hints)
RN	BFS	**28**	71
	SSSP	**24**	65
PK	BFS	**18**	42
	SSSP	**18**	41
LJ	BFS	**24**	44
	SSSP	**20**	41

Table 6.3: Average cycles spent on write task in manually implemented Swarm BFS and SSSP Bellman-Ford, with and without spatial hint attached to write task.

6.4 Performance of Hand-Tuned Low-Level Swarm Optimizations

To explore the effect of individual optimizations, we first evaluate the performance of manually implemented low-level Swarm application code for BFS and SSSP (Bellman-Ford), as described in Section 5.1. We utilize a subset of the same input graphs and a similar 64 core configuration.

6.4.1 Spatial Hint Assignment

As shown in Table 6.2, assigning spatial hints to both read and write tasks dramatically improved performance by up to 3.4x. We break down the effect of spatial hints further in Table 6.3, where we compare the average cycles spent on tasks that perform writes in BFS and SSSP. In both applications, a single task updates some in-memory value for a vertex - for example, in BFS, a vertex's parent may be updated, and in SSSP, a vertex's shortest distance is updated. Assigning spatial hints to these tasks clearly improves their performance in both applications, with up to nearly 3x faster writes,
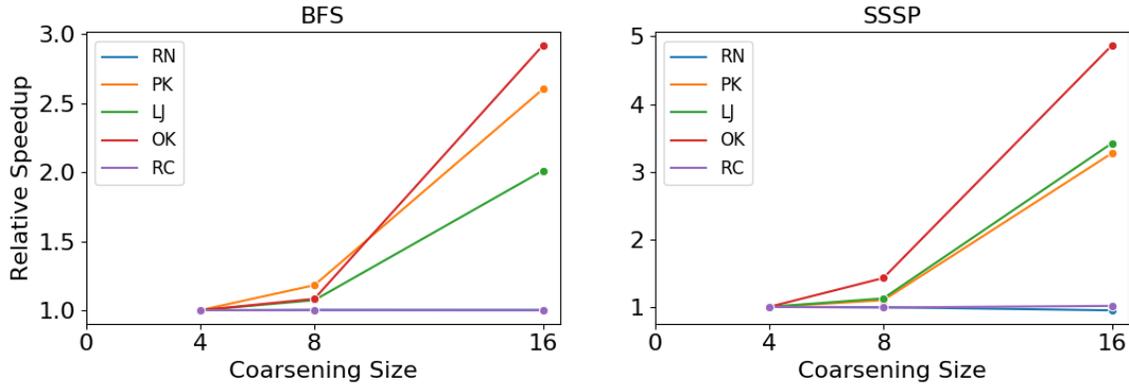
Figure 6-4: Speedup of manually implemented BFS and SSSP Bellman-Ford Swarm code using a loop coarsening factor of 4, 8, and 16 edges.

Graph	SSSP	BFS
RN	1.01	1.00
RC	1.01	0.94
PK	1.09	1.09
HW	1.06	1.05
LJ	1.06	1.06

Table 6.4: Speedup of Swarm GraphVM generated BFS and SSSP Δ-Stepping code with loop coarsening enabled (coarsened by 16 edges).

as sending tasks that modify close or same vertices to the same core incurs benefits from data locality.

6.4.2 Task Coarsening

With the manual implementations of BFS and SSSP, we also evaluated the effects of coarsening tasks to process multiple edges. In both cases, we find that coarsening generally produces speedup, especially on higher degree graphs, such as Pokec and LiveJournal. Vertices in these higher-degree graphs generally have many more edges to process, so bundling these edges together to be processed in coarser tasks can improve cache utilization. As demonstrated in Figure 6-4, application performance improves as the coarsening factor increases from 4 to 16 because the configuration cache line (64 bytes) can fit up to 16 integer values (4 bytes). Road networks have little to no improvement, as vertices in these graphs are of lower degree. Coarsening does trade off fine-grained execution for locality, so lower-degree graphs may not benefit as much from coarsening.

Clearly, specification of coarsening factor can dramatically improve performance. When enforcing this specificity in the code generated by the UGF Swarm GraphVM using scheduling options, we find similar speedups, as described in the next section.

6.5 Performance of Swarm GraphVM Optimizations

We next perform a similar evaluation of individual optimizations for code generated by the Swarm GraphVM, specifically the effect of spatial hints and coarsening, as discussed in the previous section. We compare speedups from manually implemented optimizations and GraphVM generated optimizations specified by the input schedule. Because T4 already automatically performs several optimizations during compilation, we evaluate the effect of spatial hint task boundaries and loop coarsening annotations explicitly added by the Swarm GraphVM (not automatically added by T4).

6.5.1 Spatial Hint Assignment and Task Boundaries

Spatial hints send fine-grained tasks to specific cores for execution, and explicitly added task boundaries make tasks more fine-grained, such that tasks work with more restricted sets of memory accesses. This optimization is enabled on `EdgeSetIterator` expressions by the `configLoopCoarsening` function in the Swarm scheduling language. Table 6.2 demonstrates the effect of enabling this optimization on BFS and SSSP - here, task annotations are explicitly added between reading source edge information and writing to destination vertex. As a note, T4 already automatically attaches hints to several tasks, so speedups appear less prominent when compared to the manual code speedup. Despite this, the single added task annotation still improved performance across multiple graph types by up to 17%.

We also evaluate the effect of this optimization on PageRank, where vertex update tasks are explicitly separated from vertex read tasks using the Swarm GraphVM runtime implementation of `sum_reduce`. In Table 6.5, we observe a much larger speedup, up to 5x, because update tasks in PR can be efficiently divided into extremely fine-grained read and write sections. Notably, these update tasks are also generally independent

Graph	Speedup
RN	5.37
RC	1.44
RU	2.49
PK	1.77
HW	2.99
LJ	2.50
OK	1.90
IC	1.61
TW	2.21
SW	1.21

Table 6.5: Speedup of Swarm GraphVM generated PageRank (PR) code with spatial hint option enabled.

Graph	PR	CC
PK	1.27	1.17
HW	0.92	1.71
LJ	1.05	1.22
OK	1.27	2.41
SW	1.10	1.72

Table 6.6: Speedup in Swarm GraphVM generated PageRank and CC code with loop coarsening enabled (coarsened by 16 edges).

of each other, so they may experience less aborts than in the data-driven algorithms like BFS or SSSP, which can experience more data dependences between vertex tasks. This optimization is automatically applied in the Swarm GraphVM runtime library.

6.5.2 Task Coarsening and Reordering

Tasks that iterate over edgesets can be coarsened using the scheduling option in the Swarm scheduling language. We evaluate the effect of coarsening in Table 6.6. Here, we enable the coarsening option for PR and CC, two topology-driven applications, on high-degree graphs, using the default coarsening factor of `SWARM_CACHELINE / sizeof(int)`, which is 16. As explained in Section 5.1.2, these graphs stand to benefit the most from coarsening, so we observe speedups of up to 2.4x after applying this scheduling option.

For data-driven algorithms, we also observe speedups from coarsening in power-law degree distribution graphs, as shown in Table 6.4. While less pronounced, these speedups match those produced in low-level Swarm implementation experiments, with

61

Graph	Speedup	Δ % Cycles Aborted
RN	8.00	-12.03%
RC	2.71	-65.67%
RU	4.95	-24.24%
HW	0.98	-608.78%
IC	1.24	-209.31%

Table 6.7: Speedup and reduction in aborted cycles in Swarm GraphVM generated CC code with task reshuffling enabled.

road networks experiencing little to no improvement and social graphs producing non-trivial speedups. Performance improvements are generally smaller than in the case of PR and CC - the Swarm priority queues utilized in BFS and SSSP are already well-built for these data-driven applications, as to delineate and execute fine-grained tasks across frontier rounds. Despite this, the Swarm GraphVM is still able to define the exact granularity of these tasks to find the optimal tradeoff between cache utilization and task size.

Another type of task coarsening that we evaluate in the Swarm GraphVM is the reordering of edges during an edgeset iteration. Applications that repeatedly access graphs with nodes that have high in-degrees, like Indochina or Hollywood, may experience high contention, and thus high abort costs, while processing all edges. Additionally, applications like CC, which may contest on accesses to just a few vertices, will struggle on low-degree networks like road graphs. Defining an alternative shuffling of edges prevents extremely high abort costs and yields overall performance improvements noted in Table 6.7. In the case of road graphs, we see a speedup of up to 8x, and a decrease in aborted cycles by over 65%. Both task reordering and coarsening optimizations are exposed as scheduling language options in the GraphVM, which allows programmers to easily switch between options when handling different graphs or applications.

6.5.3 Comparison to Hand-Tuned Swarm Implementations

When compared to the low-level Swarm implementations for BFS, T4-compiled code from the GraphVM is generally 1-2x slower, as seen in Table 6.8. We expect this slowdown, as T4 trades off high-level abstraction and code simplicity for less customizability, as opposed to the low-level API, which gives programmers the control

Graph	Optimized GraphVM BFS Cycles	Hand-tuned BFS Cycles (Relative Speedup)
RN	9,051,216	**4,208,216 (2.15)**
RC	63,665,217	**27,540,217 (2.31)**
PK	24,833,617	**16,659,216 (1.49)**
LJ	80,695,216	**62,313,618 (1.29)**
OK	143,978,217	**106,539,017 (1.35)**

Table 6.8: Performance of optimized Swarm GraphVM generated BFS versus manually hand-tuned Swarm BFS code, measured in runtime cycles. Relative speedup is listed in parantheses.

to write extremely optimized implementations. However, optimized generated code from the Swarm GraphVM still competes extremely well with the manually implemented version on social graphs. Given that the speedup of optimized over baseline GraphVM-generated BFS code already often reaches 10-20% (Figure 6-1), it's highly likely that with future optimizations, Swarm GraphVM generated code will reach the performance of manually hand-tuned code, if not exceed it. In other cases with larger performance gaps, there is clearly lots of potential for uncovering and exposing more optimizations, motivating future work in this domain.

Chapter 7

Conclusion

7.1   Summary

We develop UGF, a framework for generating highly optimized code for graph applications on multiple, diverse architectures. UGF achieves portability across hardware by separating hardware-independent from hardware-dependent optimizations and transformations. Using the novel GraphIR, UGF reuses hardware-agnostic compiler passes to generate a hardware-independent representation, which is lowered into hardware-specific GraphVM's. UGF also presents a new extensible scheduling language, which new hardware backends can use to configure their own scheduling optimizations.

We demonstrate the feasibility and effectiveness of UGF by developing the Swarm GraphVM, a compiler backend for the Swarm architecture. The Swarm GraphVM implements several Swarm-specific optimizations - it converts loops to fine-grained, vertex-level tasks and privatizes shared variable accesses to task-local parameters. We also expose customizable scheduling options, including spatial hint and loop coarsening configuration. Using these optimizations, we evaluate the Swarm GraphVM on 5 applications and 10 graphs, and we find that applications with fully optimized schedules achieve up to 8x speedup when compared to code generated using the same algorithm code with default schedules. Both loop coarsening and task boundaries with spatial hint assignment individually produce speedups in GraphVM generated code.

In all, UGF currently supports GraphVM's for CPU, GPU, Swarm, and the Ham-

merBlade manycore, demonstrating its ability to generate optimized code for a diverse set of hardware. UGF provides a flexible framework to leverage each architecture's unique feature set and easily support the creation of new GraphVM's.

7.2 Future Directions

In this work, we provide an example of UGF supporting a diverse set of architectures by implementing the Swarm GraphVM. Specifically with the Swarm GraphVM, we will continue to add optimizations and transformations as the Swarm architecture and programming model evolve. For instance, there is still room to explore what a "pull" traversal on Swarm may look like and how that might be optimized. In this work, only simple, single schedules were considered for Swarm, but in the future, composite (hybrid) schedules may be explored. Additionally, with the rise of other manycore and multicore architectures targeting graph applications, other GraphVM's may be proposed to support these backends. In developing these, we may find reusable passes that can help explore new optimizations for other architectures.

Appendix A

Baseline and Optimized Schedules

Algo	Baseline Schedule

PR	`SimpleSwarmSchedule sched1;`
	`program->applySwarmSchedule("s1", sched1);`
	Spatial hints disabled on `sum_reduce` operation.

CC	`SimpleSwarmSchedule sched1;`
	`sched1.configStride(STRIDE_OFF);`
	`program->applySwarmSchedule("s0:s1", sched1);`

SSSP	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(DISABLED);`
	`sched1.configLoopCoarsening(DISABLED);`
	`sched1.configSpatialHint(DISABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(PRIOQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`

BFS	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(DISABLED);`
	`sched1.configLoopCoarsening(DISABLED);`
	`sched1.configSpatialHint(DISABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(UNORDEREDQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`

BC	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(ENABLED);`
	`sched1.configLoopCoarsening(DISABLED);`
	`sched1.configSpatialHint(DISABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(UNORDEREDQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`
	`SimpleSwarmSchedule sched2;`
	`sched2.configDeduplication(DISABLED);`
	`sched2.configLoopCoarsening(DISABLED);`
	`sched2.configSpatialHint(DISABLED);`
	`program->applySwarmSchedule("s2", sched2);`
	Spatial hints disabled on `sum_reduce` operation.

Table A.1: Baseline schedules for experiments described in this thesis.

Optimal PR Schedules

Graphs with Highly Contested Nodes (HW, IC)	Other Graphs (RN, RC, RU, LJ, PK, OK, SW, TW)

`SimpleSwarmSchedule sched1;`
`sched1->configLoopCoarsening(DISABLED);`
`program->applySwarmSchedule("s1", sched1);`
Spatial hints enabled on **sum_reduce** operation.
	`SimpleSwarmSchedule sched1;`
	`sched1->configLoopCoarsening(ENABLED);`
	`program->applySwarmSchedule("s1", sched1);`
	Spatial hints enabled on **sum_reduce** operation.

Optimal CC Schedules

Road Graphs (RN, RC, RU), TW, IC	Other Power Law Graphs (LJ, PK, HW, OK, SW)

`SimpleSwarmSchedule sched1;`
`sched1.configStride(STRIDE_ON);`
`sched1.configLoopCoarsening(DISABLED);`
`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched1;`
	`sched1.configStride(STRIDE_OFF);`
	`sched1.configLoopCoarsening(ENABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`

Optimal SSSP Schedules

Large Social Graphs (TW, SW, IC), HW	Other Power Law, Road Graphs (LJ, PK, OK, RN, RC, RU)

`SimpleSwarmSchedule sched1;`
`sched1.configDeduplication(DISABLED);`
`sched1.configLoopCoarsening(DISABLED);`
`sched1.configSpatialHint(DISABLED);`
`program->applySwarmSchedule("s1", sched1);`
	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(DISABLED);`
	`sched1.configLoopCoarsening(ENABLED);`
	`sched1.configSpatialHint(ENABLED);`
	`program->applySwarmSchedule("s1", sched1);`

Optimal BFS Schedules

Graphs with Highly Contested Nodes (HW, IC)	Other Large Graphs (TW, SW)

`SimpleSwarmSchedule sched1;`
`sched1.configDeduplication(DISABLED);`
`sched1.configLoopCoarsening(ENABLED);`
`sched1.configSpatialHint(DISABLED);`
`program->applySwarmSchedule("s0:s1", sched1);`
`SimpleSwarmSchedule sched0;`
`sched0->configQueueType(UNORDEREDQUEUE);`
`program->applySwarmSchedule("s0", sched0);`
	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(DISABLED);`
	`sched1.configLoopCoarsening(ENABLED);`
	`sched1.configSpatialHint(ENABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(UNORDEREDQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`

Road Graphs (RN, RC, RU)	Other Power Law Graphs (LJ, PK, OK)

`SimpleSwarmSchedule sched1;`
`sched1.configDeduplication(DISABLED);`
`sched1.configLoopCoarsening(DISABLED);`
`sched1.configSpatialHint(ENABLED);`
`program->applySwarmSchedule("s0:s1", sched1);`
`SimpleSwarmSchedule sched0;`
`sched0->configQueueType(PRIOQUEUE);`
`program->applySwarmSchedule("s0", sched0);`
	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(DISABLED);`
	`sched1.configLoopCoarsening(ENABLED);`
	`sched1.configSpatialHint(ENABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(PRIOQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`

Optimal BC Schedules

Graphs with Highly Contested Nodes (HW, IC)	Other Graphs (RN, RC, RU, LJ, PK, OK, SW, TW)

`SimpleSwarmSchedule sched1;`
`sched1.configDeduplication(ENABLED);`
`sched1.configLoopCoarsening(ENABLED);`
`sched1.configSpatialHint(DISABLED);`
`program->applySwarmSchedule("s0:s1", sched1);`
`SimpleSwarmSchedule sched2;`
`sched1.configLoopCoarsening(ENABLED);`
`sched1.configSpatialHint(DISABLED);`
`program->applySwarmSchedule("s2", sched2);`
`SimpleSwarmSchedule sched0;`
`sched0->configQueueType(UNORDEREDQUEUE);`
`program->applySwarmSchedule("s0", sched0);`
Spatial hints enabled on **sum_reduce** operation.
	`SimpleSwarmSchedule sched1;`
	`sched1.configDeduplication(ENABLED);`
	`sched1.configLoopCoarsening(ENABLED);`
	`sched1.configSpatialHint(ENABLED);`
	`program->applySwarmSchedule("s0:s1", sched1);`
	`SimpleSwarmSchedule sched2;`
	`sched1.configLoopCoarsening(ENABLED);`
	`sched1.configSpatialHint(ENABLED);`
	`program->applySwarmSchedule("s2", sched2);`
	`SimpleSwarmSchedule sched0;`
	`sched0->configQueueType(UNORDEREDQUEUE);`
	`program->applySwarmSchedule("s0", sched0);`
	Spatial hints enabled on **sum_reduce** operation.

Table A.2: Optimized schedules for experiments described in this thesis.

Appendix B

GraphIR Nodes

Base Classes:

Class	Description
`Expr`	Representation for each expression in input algorithm code.
`Stmt`	Representation for each statement in input algorithm code.
`Type`	Representation for types of variables.

Main GraphIR nodes, with arguments listed:

Node	Arguments
`StringLiteral, IntLiteral, BoolLiteral, FloatLiteral`	`val`
`StmtBlock`	`stmts`
`ScalarType`	`type`
`ElementType`	`ident`
`VectorType`	`vector_element_type`, `range_indexset`
`VertexSetType`	`element`
`ListType`	`element_type`
`EdgeSetType`	`element`, `weight_type`, `vertex_element_type`
`ForLoopDomain`	`lower`, `upper`
`NameNode`	`body`
`ForLoopStmt`	`loop_var`, `domain`, `body`
`EdgeSetIterator`	`target`, `from_func`, `to_func`, `input_function_name`, `tracking_field`
`VertexSetIterator`	`target`, `input_function_name`, `tracking_field`
`WhileLoopStmt`	`cond`, `body`
`AssignStmt`	`left_hand_side`, `right_hand_expr`
`ReduceStmt`	`reduce_op`, `right_hand_expr`, `left_hand_side`
`CompareAndSwapStmt`	`compare_val_expr`, `right_hand_expr`, `left_hand_side`, `tracking_var`

PrintStmt	expr
IdentDecl	name, type
StructTypeDecl	name, fields
VarDecl	modifer, name, type, init_val, needs_allocation
VarExpr	var
FuncDecl	name, args, type, result, body
TensorStructReadExpr, TensorArrayReadExpr	index, target
Call	name, args
VertexSetAllocExpr	size_expr, element_type
PriorityUpdateOperatorMin, PriorityUpdateOperatorSum	new_val, old_val
LoadExpr	name
EdgeSetLoadExpr	file_name, is_weighted
VertexSetWhereExpr, EdgeSetWhereExpr	target, is_constant_set, input_func
VertexSetAllocExpr	size_expr, element_type
VectorAllocExpr	size_expr, scalar_type, vector_type
ListAllocExpr	size_expr, element_type
AndExpr, OrExpr, XorExpr, AddExpr, SubExpr, MulExpr, DivExpr	left_hand_side, right_hand_side
EqExpr	ops, operands
NotExpr	operand
NegExpr	negate, operand
IfStmt	cond, if_body, else_body
BreakStmt	
PriorityQueueType	element, priority_type
PriorityQueueAllocExpr	element_type, dup_within_bucket, dup_across_bucket, vector_function, bucket_ordering, priority_ordering, init_bucket, starting_node
UpdatePriorityUpdate-BucketsCall	priority_queue_name, lambda_name, modified_-vertexsubset_name, nodes_init_in_bucket
UpdatePriorityExternCall	input_set, priority_queue_name, lambda_name, output_-set_name, apply_function_name
UpdatePriorityEdgeCount-EdgeSetIterator	lambda_name, moved_object_name, priority_queue_-name, target, from_func, to_func, input_function_name, tracking_field
OrderedProcessingOperator	while_cond_expr, edge_update_func, priority_queue_-name, optional_source_node, graph_name
EnqueueVertex	vertex_id, vertex_frontier

Table B.1: Description of all GraphIR nodes and their arguments.

Bibliography

[1] Maleen Abeydeera, Suvinay Subramanian, Mark C. Jeffrey, Joel Emer, and Daniel Sanchez. SAM: Optimizing multithreaded cores for speculative parallelism. In *Proc. PACT-26*, 2017.

[2] Spiros N Agathos, Alexandros Papadogiannakis, and Vassilios V Dimakopoulos. Targeting the Parallella. In *Proc. Euro-Par*, 2015.

[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)*, 2015.

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proc. SC12*, 2012.

[5] Scott Beamer, Krste Asanović, and David Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *Proc. IISWC*, 2015.

[6] 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*, 2017.

[7] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Compiling graph applications for GPUs with GraphIt. In *Proc. CGO*, 2021.

[8] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 2018.

[9] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM TOMS*, 2011.

[10] Camil Demetrescu, Andrew Goldberg, and David Johnson. 9th DIMACS implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[11] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proc. SPAA*, 2018.

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

[13] Linley Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 2011.

[14] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. MultiGraph: Efficient graph processing on GPUs. In *Proc. PACT-26*, 2017.

[15] 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)*, 2019.

[16] Claire Hsu, Markus Buehler, and Anna Tarakanova. The order-disorder continuum: Linking predictions of protein structure and disorder through molecular simulation. *Scientific Reports*, 10:2068, 02 2020.

[17] Norman P. Hummon and Patrick Doreian. Computational methods for social network analysis. *Social Networks*, 12(4):273–288, 1990.

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

[19] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proc. MICRO-48*, 2015.

[20] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the Swarm architecture. *IEEE Micro*, 36(3), May-June 2016.

[21] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, August 2018.

[22] Jeremy Kepner, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, Jose Moreira, Peter Aaltonen, David Bader, and et al. Mathematical foundations of the graphblas. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2016.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[24] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

[25] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert E. Tarjan. Clustering social networks. In *Proceedings of the 5th International Conference on Algorithms and Models for the Web-Graph*, WAW'07, page 56–67, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[27] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *Proc. OOPSLA*, 2016.

[28] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Conference on Annual Technical Conference*, 2012.

[29] Carl Ramey. TILE-Gx100 ManyCore processor: Acceleration interfaces and architecture. In *Proc. HotChips*, 2011.

[30] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proc. AAAI-29*, 2015.

[31] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, 50(6), January 2018.

[32] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proc. PPoPP*, 2013.

[33] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. High-performance graph analytics on manycore processors. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015.

[34] M. B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. ISCA-31*, 2004.

[35] 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. *TOPC*, 2017.

[36] Yan Yan, Shenggui Zhang, and Fang-Xiang Wu. Applications of graph theory in protein structure identification. *Proteome science*, 9 Suppl 1:S17, 10 2011.

[37] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. T4: Compiling sequential code for effective speculative parallelization in hardware. In *Proc. ISCA-47*, 2020.

[38] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015.

[39] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with GraphIt. In *Proc. CGO*, 2020.

[40] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *Proc. BigData*, 2017.

[41] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A high-performance graph DSL. In *Proc. OOPSLA*, 2018.