Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures

Ajay Brahmakshatriya
MIT CSAIL
ajaybr@mit.edu

Emily Furst
University of Washington
eafurst@cs.washington.edu

Victor A. Ying
MIT CSAIL
victory@csail.mit.edu

Claire Hsu
MIT CSAIL
clhsu@mit.edu

Changwan Hong
MIT CSAIL
changwan@mit.edu

Max Ruttenberg
University of Washington
mrutt@cs.washington.edu

Yunming Zhang
MIT CSAIL
yunming@mit.edu

Dai Cheol Jung
University of Washington
dcjung@uw.edu

Dustin Richmond
University of Washington
dustinar@uw.edu

Michael B. Taylor
University of Washington
prof.taylor@gmail.com

Julian Shun
MIT CSAIL
jshun@mit.edu

Mark Oskin
University of Washington
oskin@cs.washington.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

***Abstract*—We live in a new *Cambrian Explosion* of hardware devices. The end of conventional processor scaling has driven research and industry practice to explore a new generation of approaches. The old DNA of architecture design, including vectors, threads, shared or private memories, coherence or message passing, dataflow or von Neumann execution, are hybridized together in new and exciting ways. Each new architecture exposes a unique hardware-level API. Performance and energy efficiency are critically dependent on how well programs can use these APIs. One approach is to implement custom libraries for each new hardware architecture and application domain. A more scalable approach is to utilize a portable compiler infrastructure tailored to the application domain that makes it easy to generate efficient code for a diverse set of architectures with minimal porting effort.**

We propose the Unified GraphIt Compiler framework (UGC), which does exactly this for graph applications. UGC achieves portability with reasonable effort by decoupling the architecture-independent algorithm from the architecture-specific schedules and backends. We introduce a new domain-specific intermediate representation, GraphIR, that is key to this decoupling. GraphIR encodes high-level algorithm and optimization information needed for hardware-specific code generation, making it easy to develop different backends (GraphVMs) for diverse architectures, including CPUs, GPUs, and next-generation hardware such as Swarm and the HammerBlade manycore. We also build scheduling language extensions that make it easy to expose optimization decisions like load balancing strategies, blocking for locality, and other data structure choices. We evaluate UGC on five algorithms and 10 input graphs on these 4 distinct architectures and show that UGC enables implementing optimizations that can provide up to $53\times$ speedup over programmer-generated straightforward implementations.

***Index Terms*—Compilers for Novel Architectures, Domain-Specific Languages, Graphs, Intermediate Representations**

I. Introduction

As we enter the twilight of Moore's Law, architectural diversity is rapidly exploding. New designs from generic parallel substrates such as manycores and dataflow engines, to highly domain-specific engines such as machine learning and graph accelerators are being researched and commercially deployed. Many of these architectures are programmable and combine well-understood techniques such as vectorization, threading, and explicit data movement in novel ways. Oftentimes, the difference between lackluster performance and dramatic speedup hinges on correctly using the particular combination of features an architecture provides. Yet it is a daunting task to do so for the wide range of hardware architectures and application domains targeted by general-purpose systems.

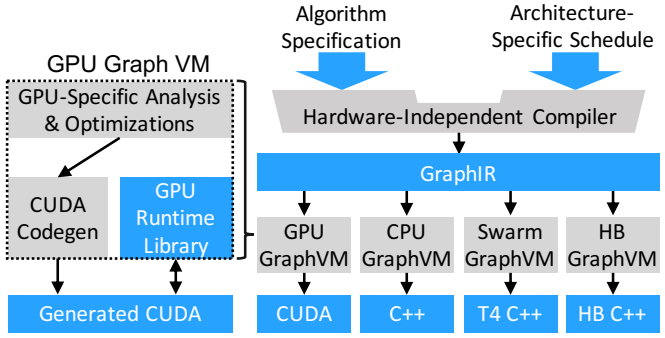


Fig. 1: The Unified GraphIt Compiler framework (UGC). GraphIR decouples the hardware-independent part of the compiler from the hardware-dependent GraphVMs. Grey blocks denote parts of the compiler, and blue blocks denote code (inputs, intermediates, libraries, or generated).

In this paper, we advocate for a novel compiler and software stack that can support this explosion in architectural diversity. We pursue a domain-specific approach, focusing on graph analytics, to enable the compiler to capture programmer intent and produce optimized implementations. We present a compiler toolchain, the Unified GraphIt Compiler framework (UGC), that targets diverse architectures while making it easy to write and compose optimizations that make use of each architecture's unique features. Recent work has developed domain-specific toolchains for deep learning and image processing [22, 70] that target CPUs, GPUs, and accelerators, showing the potential of this approach. But graphs, due to their irregularity, present a unique set of challenges for both hardware and software.

Graph processing is a crucial application domain that can benefit from hardware acceleration [6, 23, 34, 43, 87]. Graph algorithms are at the heart of many applications [16, 29, 68, 75, 90], but are notoriously difficult to optimize [59, 95]. Graph programs exhibit irregular memory access patterns that often

saturate memory bandwidth or suffer from poor utilization of hardware optimized for regular memory accesses. The diversity of graph applications and input graphs, combined with the unique features of different architectures, makes it hard to program high-performance and portable graph applications. For example, when processing smaller graphs on CPUs, exploiting the cache hierarchy and out-of-order execution are key. On GPUs, which have two orders of magnitude more compute and memory bandwidth [65], structuring the code to exploit data parallelism and block-oriented memory accesses are key. On task-based architectures such as Swarm [43], exploiting speculation and fast inter-task synchronization is critical. Finally, on manycore architectures such as HammerBlade, which have hundreds to thousands of small general-purpose processor tiles [5, 25, 33, 71, 84], it is critical to efficiently use fast software-managed scratchpad memory.

Choosing the right level of abstraction for the intermediate representation is critical to simplify code generation for the above diverse architectures and to expose optimizations opportunities. To achieve these goals, UGC introduces a new domain-specific intermediate representation, the Graph Intermediate Representation (GraphIR), to encode hardware-independent optimizations and to serve as a high-level interface to different hardware backends.

UGC is built on top of the GraphIt domain-specific language (DSL) [15, 93, 95], which decouples the algorithm from the performance optimizations (schedules) for graph programs. UGC uses a new scheduling language that combines load balancing, edge traversal direction, active vertex set creation, active vertex set ordering, kernel fusion, explicit data movement, and fine-grained task splitting, among other optimizations. Figure 1 depicts the overall compilation flow. First, various analyses and lowering passes generate GraphIR. Then, GraphIR is lowered into code for different architectures using an architecture-specific Graph Virtual Machine (GraphVM), which performs hardware-specific transformations and code generation.

This paper makes the following contributions:

- A compiler framework with a novel and carefully designed intermediate representation, GraphIR (Section III-B); hardware-independent passes; and hardware-specific GraphVMs to generate fast code on diverse architectures.
- A novel extensible scheduling language (Section III-D) that allows programmers to explore the optimization spaces of different hardware platforms.
- Implementations of four GraphVMs that can generate efficient code for CPUs, GPUs, Swarm, and the HammerBlade Manycore (Section III-C).
- Evaluation of code generated by the four GraphVMs, which shows up to 53× speedup over user-supplied baseline code.

This paper also provides insights on techniques for building portable compilers targeting very different architectures for a specific application domain.

II. Background

The performance of graph processing depends on optimizations to mine for locality within sparse data structures, to minimize high-cost memory accesses and synchronization, and to balance load across parallel threads [11, 59, 86]. Unfortunately, the structure of graphs varies widely, as does the work across iterations of an algorithm. As a result, the optimal approach can change not only across graphs, but also across iterations [10, 60]. This makes graph programs notoriously difficult to optimize on any architecture.

To make matters worse, modern architectures employ a wide range of hardware features to exploit parallelism and achieve high throughput: threads, vectors and warps, tasks, task or instruction speculation, memory consistency models, cache coherence, variants of atomic operations, data movement engines, and scratchpads (to name a few). Combinations of these features produce exponentially many architectural variations, each with different performance characteristics. Each architecture has a different low-level language, compiler, and runtime that exposes these features, and implementations must be cognizant of these features and their implications.

Domain-specific languages (DSLs) for graph processing abstract the complexity of modern architectures [95] and the dynamic challenges of graph structure. DSLs have been used to abstract hardware in domains like machine learning [22, 53], image processing [70], networking [14, 51], tensor algebra [50], and bioinformatics [74], or sometimes combining a few domains [18, 72]. An ideal DSL for graph processing would facilitate algorithm expression and abstract away architectural details to provide good performance on a wide range of applications and architectures.

A. GraphIt Domain-Specific Language (DSL)

GraphIt [15, 93, 95] is a domain-specific language for graph applications that decouples the algorithm specification and computation schedule. This enables GraphIt to generate high-performance code with optimizations tailored for diverse graph inputs from a single portable algorithm specification.

```
1  ...
2  func toFilter(v : Vertex) -> output : bool
3    output = (parent[v] == -1);
4  end
5  func updateEdge(src : Vertex, dst : Vertex)
6    parent[dst] = src;
7  end
8  func main()
9    ...
10   #s0# while(frontier.getVertexSetSize() != 0)
11     #s1# var output : vertexset{Vertex} =
12       edges.from(frontier).to(toFilter).
13         applyModified(updateEdge, parent, true);
14   ...
15   end
16   delete frontier;
17 end
```

Fig. 2: Algorithm specification for Breadth-First Search (BFS) in GraphIt.

To concretely show the benefits of GraphIt's approach, Figure 2 shows the algorithm specification for Breadth-First Search (BFS) in GraphIt. This code only describes the computation to be performed. Lines 2 and 5 define functions for filtering vertices and updating edges. Line 13

Hardware features	CPU	Swarm	GPU	HammerBlade
Parallel Execution Model	Threads	Ordered Tasks	SIMT / SIMD	SPMD
Number of Processors	∼ 100	∼ 100	∼ 100K (Threads)	∼ 1000
Speculation	Instruction-Level	Instruction- & Task-Level	No	No
Memory Latency Hiding	OoO execution, SMT	OoO execution, SMT	Multithreading	Non-Blocking Memory Ops.
Synchronization support	Atomics	Coherence-Enforced Ordering	Atomics, Barriers	Atomics
Addressable Memory (GB)	∼ 1000	∼ 1000	∼ 40	∼ 50
Shared on-chip Memory	Coherent L3	Coherent L3	L2	Globally-Partitioned LLC
Core-local data Memory	Coherent L1,L2	Coherent L1,L2	L1 (No Coherence)	Scratchpad (SW Coherence)
On-Chip Storage per Thread	∼ 1MB	∼ 1MB	∼ 100B	4KB

TABLE I: Summary of parallel architectures studied in this paper. Two of the four architectures are shown in Figure 3.

calls the `edgeset.applyModified` operator, which uses these functions to specify which edges are to be processed and what computation is to be performed on each edge. The algorithm specification does not specify the loop nests or the iteration order; this is specified in the schedule. This separation makes it possible to generate different implementations suitable to the algorithm and graph input.

UGC uses exactly the same algorithm language as GraphIt, enabling us to reuse the source code written for various applications. The high-level design of the operators also makes it easy for UGC to target very different architectures. For example, the `edgeset.applyModified` operator can be easily mapped to architectures that have specialized units for traversing edges in parallel. We extend the scheduling language to fit the optimizations of different architectures. Examples are shown in Figure 6.

B. Parallel Architectures

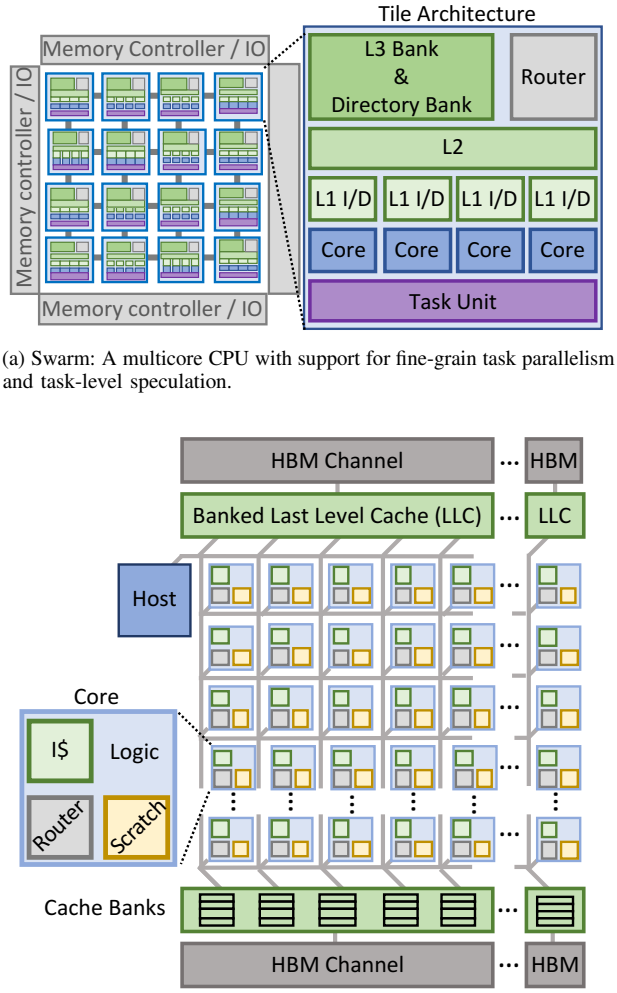
In this work, we target the four parallel architectures shown in Table I. These architectures are built with a diverse set of hardware features that expose different forms of parallelism, latency hiding techniques, and synchronization. These architectures require significantly different optimization strategies and pose unique challenges for UGC. We briefly explain each architecture below, and the challenges that they present when compiling graph programs.

1) Multicore CPU

A multicore CPU has cores optimized for single-thread performance, with prefetching and a multi-level cache hierarchy. To hide latency, each core supports speculative out-of-order execution as well as simultaneous multithreading. CPU software exposes explicit parallelism through threads provided by a multithreaded runtime [13, 19]. CPUs perform well on graph applications that provide limited parallelism, high locality, or predictable memory access patterns [11]. The large memory capacity also means that CPUs outperform other systems on multi-terabyte graphs [28].

2) GPU

GPUs provide massive parallelism through a SIMT programming (SIMD execution) model, where arithmetic units are vectorized and use predication to handle divergent control flow. GPUs use multithreading with many hardware thread contexts to hide memory latency. GPUs are suitable for graph applications with massive parallelism that exhibit regularity in graph structure, memory access pattern, and limited control flow. Applications such as PageRank [66] or less-sparse graph workloads that map well to existing linear algebra libraries [45, 89] can exploit massive memory bandwidth with coalesced memory accesses. GPUs perform poorly on applications that suffer from control divergence [88], load-imbalance [77], or too many sparse memory accesses [1]. Finally, GPUs function best on graphs that fit in device global memory.



(a) Swarm: A multicore CPU with support for fine-grain task parallelism and task-level speculation.

(b) HammerBlade Manycore Pod: Cores with software-managed scratchpads operate independently in a partitioned, globally-addressable memory space.

Fig. 3: Architectural overview of two of the parallel graph processing architectures studied in this paper.

3) Swarm

Swarm augments a CPU with support for fine-grained task parallelism [43]. Swarm can achieve order of magnitude

improvements in scalability over conventional CPUs and GPUs on some graph algorithms by using dedicated hardware task queues and speculative execution to distribute tasks across hundreds of cores [42, 44, 81].

Swarm's execution model uses order as the main synchronization primitive. Swarm programs consist of tasks. Each task can read and write arbitrary memory and spawn children tasks. Each task is given a timestamp when it is spawned, which must be greater than or equal to its parent's timestamp. Swarm guarantees that tasks appear to run atomically and in timestamp order, hiding the effects of concurrency from software. Under the hood, Swarm hardware executes tasks in parallel and out of order. To preserve ordered semantics, tasks execute speculatively and the coherence protocol is extended to detect order violations. Upon a violation, the offending task is aborted and re-executed. Fig. 3a shows how Swarm adds a task unit near the cores of each chip tile. These distributed task units perform asynchronous, high-throughput task dispatch and task commit, efficiently supporting tasks as short as a few instructions. These tiny tasks can be selectively aborted or serialized with compiler-generated hints, exposing unique tradeoffs as optimizations must balance task overheads, parallelism, and the costs of aborts and re-executions.

This execution model is a natural fit for priority-based or iterative algorithms, where each task can be assigned a timestamp based on its priority or iteration number. Swarm's speculative execution uncovers more parallelism than CPUs and GPUs by executing tasks with different timestamps in parallel. Swarm can be programmed in C++ using the T4 compiler [91], but the key challenge is in appropriately dividing the computation into tiny tasks to exploit parallelism and minimize abort costs.

4) HammerBlade Manycore

Manycore architectures provide thread-level parallelism and flexibility with hundreds to thousands of general-purpose cores [5, 25, 33, 71, 84]. We target the HammerBlade Manycore with hundreds of independent cores. The cores have a scalar pipeline, low-latency software-managed scratchpad memory, and support integer, floating-point, and atomic instructions. The cores communicate over the memory-mapped 2-D mesh Network-on-Chip. Cores can issue many non-blocking memory requests to exploit pipeline parallelism and hide memory latency. In addition to the scalar cores, there is an on-network host processor that manages execution. Figure 3b presents an architectural diagram of the HammerBlade Manycore.

The HammerBlade Manycore memory hierarchy requires software to make choices to efficiently exploit memory parallelism and trade off between latency and capacity. The memory hierarchy has four levels: core-local scratchpad, inter-core scratchpad(s), banked Last Level Cache (LLC), and High-Bandwidth Memory (HBM) [41, 56]. Core-local scratchpad, remote scratchpads, cache, and other network locations are mapped to non-intersecting regions of a core's address space to give software explicit control over data movement. Scratchpads offer low-latency storage and are explicitly managed by software threads on the cores. Multiple independent HBM channels service pipelined memory requests from the LLC. Cache banks map to exclusive memory ranges of the HBM address space. Consequently, the HammerBlade Manycore exposes a PGAS-like memory model that is coherent by construction.

The HammerBlade Manycore provides a kernel-centric programming abstraction, similar to CUDA. Kernel code is written from the perspective of a single thread executing on a core. Multiple cores are aggregated into rectangular *groups* to execute kernels. Cores in a group communicate explicitly through global memory or operations on remote and local scratchpads. Cores executing within a group synchronize using explicit barrier primitives. Kernel execution and scheduling is managed through runtime software on the tightly-coupled host processor. This provides a SPMD-like execution model.

Manycore architectures are well suited for graph applications with high parallelism, and random memory access patterns. Unlike GPUs, the independent cores are not slowed by control-flow divergence. Independent HBM channels can service multiple memory accesses simultaneously. The key challenges are to use non-blocking loads to hide latency, to exploit thread-level and memory level parallelism, and to balance work between independent threads of execution.

III. Compiler Design

Choosing the right abstraction for the intermediate representation is critical to simplify code generation for diverse architectures and to expose optimization opportunities. To achieve these goals, we designed GraphIR, a novel intermediate representation. We studied the features of varied architectures to identify the right level of abstraction with enough expressiveness to capture algorithmic details from the graph domain. For example, instead of low-level loop nests, GraphIR has operators for iterating over a set of vertices, or edges incident to a set of vertices. These operators can be directly mapped to thread hierarchies or manycore tiles on architectures such as GPUs and the HammerBlade Manycore, without needing to lift computations from loop nests. GraphIR also avoids making assumptions about the concrete representation of data structures. This allows different architectures to choose various implementations for vertex sets depending on the available memory, bandwidth, and other tradeoffs. For Swarm, the compiler can even eliminate the use of software work queues for vertex sets, instead mapping the operations on vertex sets to hardware tasks. This section explains how GraphIR enables building GraphVMs with these specialized optimizations.

A. Hardware-Independent Transformations

GraphIR has a dual goal of offering flexibility while allowing for maximum code reuse. Even though GraphIR's main goal is to support specialization and optimizations unique to each hardware backend, a large part of the compiler infrastructure, including analysis and transformation passes, is target-agnostic. Specifically, UGC adapts the domain-specific transformations from the GraphIt DSL compiler [15, 93, 95], such as dependence analysis to insert atomics in the user-defined functions (UDFs), liveness analysis to find frontier memory reuse opportunities,

and other transformations to UDFs for traversal direction, parallelism, and data structure choices. These hardware-independent transformations and analyses are performed on the GraphIR before it is passed to the GraphVMs for code generation. These passes can access scheduling language inputs (Section III-D). These passes also add metadata to the GraphIR for the GraphVMs to use during code generation. Section III-C shows how the bulk of the frontend and the hardware-independent compiler are reused by all four GraphVMs that we implement.

B. GraphIR

One of the main contributions of this paper is the GraphIR intermediate representation that decouples the algorithm specification and hardware-independent optimizations from hardware-specific optimizations. Like LLVM IR, GraphIR is an in-memory representation of a program that allows optimizations through IR-to-IR transformations before final code generation. This design enables us to build reusable program analyses, transformations, and lowering passes shared across different hardware platforms, reducing the effort needed to support a new backend (GraphVM) in UGC. However, unlike LLVM IR, GraphIR uses a high-level domain-specific representation that facilitates more powerful and flexible optimizations.

GraphIR is composed of variables, functions, and instructions. Each variable, function, or instruction carries both arguments and metadata, as shown in Table II. Arguments capture all of the information derived from the algorithm specification and is required for correctness of the generated code. Metadata captures information related to the performance optimizations, and hardware backends can choose to ignore these or add new ones specific to their hardware. GraphIR's metadata can be manipulated with an API that includes two functions: `setMetadata<T>(std::string label, T val)` and `T getMetadata<T>(std::string label)`, where `T` is any C++ type (including other GraphIR nodes). Because this API allows arbitrarily many string labels, metadata can easily stack without having to change GraphIR base class definitions. This metadata API is the primary way in which GraphVMs extend GraphIR nodes for hardware-specific optimizations.

To perform hardware-specific transformations and code generation, each backend implements an abstract machine (GraphVM) to optimize and run GraphIR, similar to the Java VM or LLVM. Section III-C provides details on GraphVMs.

Operators and data types are designed in an implementation-agnostic way to make it easy for the GraphVM developer to pick the right data structure and choice of mapping computations to various hardware units. The two most important instructions in GraphIR are the `EdgeSetIterator` and `VertexSetIterator` instructions, shown in Table II. `EdgeSetIterator` iterates through all or a subset of the edges of a graph and invokes a function on each edge. The arguments of `EdgeSetIterator` specify the graph (`input_graph`), input frontier vertexset (`input_vset`), output frontier vertexset (`output_vset`), and the user-defined function that works on the edges (`apply_f`). These arguments are derived from the operators in the algorithm specification. The instruction also has metadata to generate optimized implementations, such as choosing the input/output

```
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<format=SPARSE>(output_frontier,
               dst)
6      }, {})
7  })
8  Function main (int32_t argc, char* argv[], {
9      ...
10     WhileLoopStmt<needs_fusion=true>(VertexSetSize(
           frontier), {
11         EdgeSetIterator<requires_output=true,
12             can_reuse_frontier=true,
13             direction=PUSH,
14             is_edge_parallel=true>(
15             edges, frontier, output, updateEdge,
                   toFilter),
16         AssignStmt(frontier, output)
17     }),
18 })
```

Fig. 4: Optimized GraphIR generated by the compiler for the BFS algorithm given a schedule that enables kernel fusion. This text representation is generated by pretty printing the GraphIR, which is an in-memory data structure. A backend developer can manipulate GraphIR with the UGC API.

frontier representations, edge traversal direction, deduplicating the output frontiers, or generating specialized code if the edge set representation is dense. `VertexSetIterator` iterates over the vertices in a frontier, and similarly has arguments and metadata for optimizations. Apart from these key instructions, GraphIR has instructions for data structure allocation both on the host and on the device, general arithmetic and reductions, and program control flow.

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

Figure 4 shows the pretty-printed GraphIR for the BFS algorithm input from Figure 2. Table II explains each of the GraphIR operators and types used in this example. Line 11 shows the key `EdgeSetIterator` GraphIR node. This node contains arguments such as the graph to iterate on, the input and output frontiers, the function to apply on each edge, and the source and destination filters. This operator also has metadata attached to it (shown in <>). For example, the `can_reuse_frontiers` is the result of the frontier reuse analysis pass. As shown in Table III, the result of this analysis is used by the GPU, Swarm, and HammerBlade Manycore GraphVMs. The `EnqueueVertex` node is another GraphIR node that has metadata, in this case for the representation of the frontier to enqueue to (Line 5). The code in Figure 4 is just the pretty-printed version of the in-memory GraphIR data structure.

The BFS example also shows the `updateEdge` user-defined function (UDF) that `EdgeSetIterator` applies to each edge. Line 3 shows that the high-level compiler inserted

GraphIR Types

Type	Description
Vertex	Type to represent an individual vertex in the graph.
Edge	Type to represent a single edge in the graph.
EdgeSet	Graph data type. Can be weighted or unweighted. Can have `COO` or `CSR` representation.
VertexSet	Type to hold a set of vertices. Can have `SPARSE`, `BITMAP` or `BOOLMAP` representation.
Function	Top level function definition type. Functions can be annotated as `DEVICE`, `HOST` or both.
VertexData	A property of a basic type (`float`, `int`, `long`...) associated with each vertex in the graph. Can be stored as array of struct or struct of arrays.
PrioQueue	Type to represent queues that hold vertices to be processed based on some priority.
FrontierList	Type to hold a list of `VertexSet`s.

GraphIR Instructions

Instruction	Arguments	Metadata
`VertexSetIterator`	**VertexSet** in_set **Function** apply_f	`bool` is_all_verts `bool` is_parallel
`EdgeSetIterator`	**EdgeSet** input_graph **VertexSet** input_vset **VertexSet** output_vset **Function** apply_f	`bool` is_all_edges, requires_output, apply_deduplication `bool` can_reuse_frontier, is_edge_parallel `DirectionType` direction: *PUSH* or *PULL* `VertexSetRepresentation` output_representation `VertexSetRepresentation` pull_input_frontier `PriorQueue` queue_updated
`EnqueueVertex`	**VertexSet** output **Vertex** to_output	`VertexSetRepresentation` output_format
`CompareAndSwap`	**TensorExpr<BasicType>** expr *BasicType* old_value *BasicType* new_value	`bool` is_atomic
`WhileLoopStmt`	**bool** condition	`bool` needs_fusion `List<Variables>` hoisted_vars
`UpdatePriorityMin` `/Sum`	**Vertex** to_update **int** update **PrioQueue** Q	`bool` needs_atomic
`ListAppend`	**FrontierList** list **Output** to_append	`bool` to_destroy
`ListRetrieve`	**FrontierList** list **VertexSet** output	`bool` needs_allocation
`VertexSetDedup`	**VertexSet** to_process	
`ReductionOp`	**TensorExpr<BasicType>** expr **ReductionType** type *BasicType* val	`bool` is_atomic

TABLE II: Key data types and IR nodes in GraphIR. The table also shows the arguments and metadata associated with each IR node. This metadata is attached by the hardware-independent part of the compiler. GraphVMs can add more metadata as part of their respective passes. The basic arithmetic and control flow IR nodes are not shown here for brevity.

a `CompareAndSwap` after dependence analysis with metadata `is_atomic=true`, indicating that this operation requires appropriate synchronization after lowering. Different backends (GraphVMs) implement the `CompareAndSwap` instruction differently: CPUs use compare-and-swap hardware instructions; GPUs use warp shuffling for cheap communication among threads; and the Swarm GraphVM ignores `is_atomic` metadata because Swarm hardware always executes tasks atomically.

GraphIR also facilitates hardware-specific optimizations. For example, the GPU GraphVM can fuse multiple kernel launches into a single kernel launch, as Section III-C2 will explain. The GPU GraphVM extends the metadata of `WhileLoopStmt` with a `needs_fusion` flag, and sets it to `true` in a hardware-specific pass to indicate that the schedule has prescribed fusing all of the operator calls inside the loop into a single kernel.

The right level of abstraction and the support for extending GraphIR with metadata makes GraphIR an ideal representation for accomodating hardware-specific optimizations in UGC.

C. GraphVM

The Graph Virtual Machine (or GraphVM) is an abstract machine that executes the target-independent GraphIR. Each backend developer implements a GraphVM tailored to their architecture that includes hardware-specific passes and code generation. The UGC framework provides all of the required tools to build diverse optimization passes including APIs to access GraphIR nodes and scheduling objects attached to them, a set of reusable passes that can be enabled depending on whether the hardware benefits from it, and common routines to aid code generation. GraphVMs for different architectures can be very diverse. Each GraphVM developer can implement it as an interpreter that directly consumes and executes GraphIR or as a combination of transformation and code generation passes. The developer can also choose to move complexity between the generated code or the runtime library, as we discuss next. As Figure 1 shows, a typical GraphVM has the following parts:

- Hardware-dependent analyses and transformation on GraphIR using hardware-specific scheduling information.
- Code generation for the target device and host (if applicable).
- Runtime library and backend compiler infrastructure to execute the generated code.

As shown in Table III, UGC provides a library of analysis and transformation passes that GraphVMs reuse or specialize, easing the development of new backends. We now discuss our our GraphVMs and their hardware-specific optimizations.

	Module	Base version	CPU	GPU	Swarm	HammerBlade
Frontend	Algorithm parser & AST definitions	**10,900**	0	0	0	0
	Schedule language functions	**136**	306	385	524	89
Hardware-Independent Compiler	Frontier Reuse Analysis	**125**	Not used	0	0	0
	Property analysis/Atomic insertion	**536**	0	0	Not used	0
	Ordered Processing Lowering	**386**	0	120	0	0
	Other Lowering Passes	**4,171**	0	0	0	0
GraphVM	Ordered Processing Specialization	**104**	0	Not used	0	Not used
	Kernel Fusion	276	Not used	0	Not used	Not used
	Code Generator	-	3,843	1,874	959	2,282
	Runtime Library	-	10,385	2,470	156	1,127

TABLE III: Lines of code for modules of UGC. Modules reuse code through object-oriented programming patterns, so lines of code are divided between base modules and lines added in GraphVMs. Each GraphVM may use a base pass as-is, add lines for hardware-specific optimizations, or simply not use the pass. Lines of code in **bold** are used by multiple GraphVMs.

1) Multicore CPU GraphVM

The CPU GraphVM has all of the CPU-specific passes from the original GraphIt compiler to implement optimizations specific to CPUs, such as edge-based and vertex-based traversals, different representations for priority queue data structures, cache and NUMA optimizations, vertex data array of struct and struct of array transformations, among others. The code generated from our CPU GraphVM is comparable to the code generated from the original GraphIt compiler, thus maintaining the state-of-the-art performance demonstrated in GraphIt [93, 95].

2) GPU GraphVM

The GPU GraphVM generates high-performance host and device CUDA code tuned for different generations of GPUs. Our implementation of the GPU GraphVM implements all of the optimizations in the GPU version of GraphIt [15], but in such a way that they can easily be integrated with the rest of the infrastructure by means of GraphIR. The GPU GraphVM makes use code generation as well as a large runtime library to offload some of the complexity of code generation. We provide examples of both techniques below.

Load-balancing runtime library. The GPU GraphVM [15] implements many load-balancing strategies to trade off utilization, synchronization costs, and work efficiency. Since the logic of assigning edges to threads is largely independent of the actual computation to be performed, load-balancing implementations can be cleanly moved to a set of template library functions. This not only simplifies code generation, but also makes it easier to add more load-balancing techniques.

Code generation for kernel fusion. Kernel fusion is an important optimization for road graphs because it amortizes kernel launch overheads for applications where there is very little work in each iteration [67]. The kernel fusion optimization is implemented entirely in the GPU GraphVM as a series of passes. A preliminary pass identifies the loops to be fused and all of the variables that the body of the loop uses from the main function. The first pass in the code generation then generates the actual `__global__` kernel to be launched on the GPU. The code generation pass inserts appropriate CUDA API calls to copy state between the host and device. Since the fused kernel has a fixed number of threads, the code generator also generates some outer loops to simulate the work of more threads and inserts `grid_sync()` calls for synchronization. Finally, a pass generates appropriate calls to launch a single GPU kernel instead of a separate kernel for each step in each iteration of the loop. Table III shows how this GPU-specific pass is a very small fraction of the total lines of code. This demonstrates that the design choices in GraphIR and GraphVMs significantly reduce the effort required to support unique hardware optimizations.

The GPU GraphVM also implements other optimizations, such as EdgeBlocking and fused vs. unfused frontier creation.

3) Swarm GraphVM

The Swarm architecture relies on speculative execution of tasks to extract parallelism and make applications scale to a large number of cores. Tasks are executed out of order but are aborted when memory dependencies are violated, thus ensuring correctness. However, repeated aborts are undesirable because they result in wasted work. Therefore, the Swarm GraphVM focuses a great deal on eliminating false dependencies between memory accesses. Figure 5 shows the code generated by the Swarm GraphVM for the BFS algorithm.

From vertex sets to tasks. One of GraphIR's main data types is the `VertexSet`, which holds the current set of active vertices. This active set is read from on every round and written to for the next round. Storing the active vertex set in memory introduces data dependencies (e.g., reuse of memory across rounds, or between updates to in-memory tail pointers or `size` variables) that prevent Swarm from obtaining more parallelism by speculating across many rounds. These data dependencies are spurious, because the insertion of distinct vertices should actually be independent. We solve this problem through a pass in the Swarm GraphVM that replaces the enqueuing of a vertex ID to a `VertexSet` with a task spawn. The body of this task is the operation that we would perform with the vertex after dequeuing it. The timestamp of the task is set based on the round in which the vertex would be dequeued. This way, while Swarm's execution model guarantees tasks from one round appear to execute before any task from the next round, tasks from different rounds can execute speculatively in parallel without false dependences arising from storing the `VertexSet` in memory.

Figure 5 shows a lambda passed to `for_each_prio` to indicate what action should be taken per element in the frontier. The body of the lambda calls `push` to spawn tasks that will execute the lambda on vertices at later timestamps.

This approach generalizes to priority-based algorithms like Δ-stepping as well, where task timestamps are set based on priorities.

From shared to private state. Some applications have shared variables that are updated periodically. For example, in the forward pass of BC there is a variable updated once per round to track the region of the output data structure that visited vertices are recorded to. If all parallel tasks access this single variable, data dependencies on the updates to this variable would prevent speculation across rounds. To address this, the Swarm GraphVM passes a private copy of this value to each task that needs it, and updates are performed in a functional style before passing these values to any task spawned for the next round. By avoiding updates to any copy of the variable shared by multiple parallel tasks, this pass eliminates unnecessary dependences and unlocks more speculative parallelism.

Fine-grained splitting and spatial hints. When a dependence is violated, the Swarm hardware must roll back and re-execute the work done by the offending speculative task. It is important to minimize this wasted work. We add a pass in the Swarm GraphVM that helps the hardware schedule tasks in a way that reduces both the number of aborts and the cost of each abort. Swarm's T4 compiler tries to assign spatial hints to each task based on the memory locations that it accesses, but it can do this only for tasks that do not access disparate memory addresses [91]. Line 4 in Figure 5 shows how the GraphVM adds an annotation to instruct T4 to split the subsequent block of code into a subtask that accesses a single memory address. This lets T4 dispatch these subtasks to chip tiles according to the cache line that they access. As a result, accesses to a given cache line are all executed within one chip tile, where hardware can selectively serialize tasks that access the same cache line, reducing the likelihood of aborts [42]. These fine-grained subtasks are also cheaper to roll back and re-execute if they are aborted, reducing the cost of aborts. Additionally, the GraphVM exploits domain knowledge about the loops iterating over constant edge arrays to strike a balance between the cost of aborts and the cost of spawning additional tasks, by generating annotations that help the backend compiler schedule memory access instructions.

4) HammerBlade Manycore GraphVM

The HammerBlade Manycore GraphVM produces parallel C++ code targeting the HammerBlade Manycore architecture described in Section II-B4. The code produced by this GraphVM is separated into sequential host code and parallel device code. The sequential host code handles initialization and coordination, while the parallel device code executes the body of the graph algorithm. The HammerBlade Manycore GraphVM implements optimizations and GraphIR transformations that target the manycore architecture and its memory hierarchy. Similar to the GPU GraphVM, the HammerBlade Manycore GraphVM also provides extensive host and device runtime libraries to simplify code generation.

Atomics. Similar to a GPU, atomics on the HammerBlade Manycore are expensive operations. On the manycore, atomic operations in global memory are implemented using lock data structures. The HammerBlade Manycore GraphVM leverages the atomics pass from the GPU GraphVM to determine where atomics are necessary. If an atomic operation is determined to be necessary within a kernel, initialization code for the necessary locks is also inserted into the host code.

```
1  frontier.for_each_prio([](int round, int src) {
2    for (int edgeID: neighbors(src)) {
3      int dst = edgeDst[edgeID];
4      #pragma task hint(&(parent[dst]))
5      {
6        if (parent[dst] == -1) {
7          parent[dst] = src;
8          push(round + 1, dst);
9        }
10     }
11   }
12 });
```

Fig. 5: Code generated by the Swarm GraphVM for BFS.

Blocked access optimization. The HammerBlade Manycore GraphVM implements an optimization that utilizes the software managed scratchpads on each core. We call this optimization the blocked access method. This method aims to reschedule long-latency requests to main memory in parallel by formatting work items into blocks. Cores iterate over their assigned blocks, prefetching the entire block at once. Block data is stored in the core's scratchpad memory, repurposing it as a software-managed L1 cache. The HammerBlade Manycore GraphVM determines which elements can be safely read into scratchpad memory without requiring synchronization between cores at the end of processing.

Alignment-based partitioning. Memory-level parallelism and workload partitioning are very important to achieve high performance on the HammerBlade Manycore. We propose an alignment-based partitioning method that aims to improve memory system performance. In this method, cores work on smaller work blocks of vertices that better align with cache lines in the LLC, increasing effective memory access bandwidth. This optimization utilizes a similar partitioning scheme to the blocked access method, but does not use the core's scratchpad memory. This is due to the observation that, for some graph workloads, the cost of loading data into scratchpads outweighs the benefit of low-latency scratchpad accesses. Graph vertices are split into V/b work blocks, where b is the number of vertices contained in each work block, and V is the total number of vertices in the graph. We select b to be a multiple of the cache line size. Cores work on these blocks until all work blocks have been processed. Because this reduces the size of the active vertex set that each core is working on, we are able to increase the cache hit rate and reduce cache line contention.

D. Extensible Scheduling

One of the main features of UGC is that it decouples the algorithm input from the optimization schedules. This way, the programmer or an autotuner [7] can generate different variants of the same algorithm tailored to specific graph inputs simply by supplying different schedules. Since different GraphVMs support different optimizations, we build a new scheduling

Abstract Class	Description
SimpleSchedule	Hardware-independent abstract class for Simple schedule objects.
Function	Description
getParallelization	Get the parallelization scheme of the schedule (VERTEX_BASED or EDGE_BASED).
getDirection	Get the direction of traversal of edges. Can be PUSH or PULL.
getPullFrontier	Get how the next frontier will be created. Can be BOOLMAP or BITMAP.
getDeduplication	Get whether explicit deduplication should be performed on the output frontier.
getDelta	Get the delta value to use when creating buckets in PriorityQueue.

TABLE IV: Description of the SimpleSchedule type and some associated virtual functions.

Abstract Class	Description
CompositeSchedule	Hardware-independent abstract class for hybrid schedule objects (schedule that changes based on runtime value).
Function	Description
getFirstSchedule	Get the first schedule object within this hybrid schedule.
getSecondSchedule	Get the second schedule object within this hybrid schedule.

TABLE V: Description of the CompositeSchedule type and some associated virtual functions.

language for each target. These scheduling languages have essential features for optimizations on their respective targets.

One of the challenges with this approach is that the hardware-independent part of UGC now has to deal with different scheduling languages for the parameters that it needs. For example, the dependence analysis to insert atomics in the UDFs at least needs to know if the parallelization is vertex based or edge based and if the traversal direction is PUSH or PULL.

To address this problem, we use object-oriented programming techniques to enable the hardware-independent part of UGC to query the information that it needs from various scheduling representations. The scheduling language input is stored internally as scheduling objects attached to program nodes. UGC creates an abstract interface with virtual functions for all of the information that the hardware-independent compiler needs. We implement new scheduling object classes for each GraphVM by inheriting from this abstract interface. These new classes have members and functions to configure various scheduling options specific to optimizations supported for their GraphVMs. These classes implement the virtual functions to provide the hardware-independent part of UGC with the information that it needs. Tables IV and V describe these abstract scheduling classes with the virtual functions to query information, such as direction and parallelization type.

Figure 6 shows example scheduling inputs for the BFS algorithm for different GraphVMs. The HammerBlade schedule example shows hybrid traversal with cache-aligned load balancing, while the Swarm example enables transformations for consecutive frontiers into a priority queue and breaks down updates into smaller tasks.

Figure 6a shows a use of the CompositeGPUSchedule class,

```
SimpleGPUSchedule sched1;
    sched1.configDirection(PUSH);
    sched1.configFrontierCreation(FUSED)
SimpleGPUSchedule sched2;
    sched2.configDirection(PULL, BITMAP);
    sched2.configFrontierCreation(UNFUSED_BITMAP);
CompositeGPUSchedule comp1 (INPUT_SET_SIZE, 0.15, sched1, sched2);
program->applyGPUSchdule("s0:s1", comp1);
```

(a) GPU schedule for BFS.

```
SimpleHBSchedule sched1;
    sched1.configLoadBalance(ALIGNED);
    sched1.configDirection(HYBRID);
program->applyHBSchedule("s0:s1", sched1);
```

(b) HammerBlade Manycore schedule for BFS.

```
SimpleSwarmSchedule sched1;
    sched1.configDirection(PUSH);
    sched1.taskGranularity(FINE_GRAINED);
    sched1.configFrontiers(VERTEXSET_TO_TASKS);
program->applySwarmSchedule("s0:s1", sched1);
```

(c) Swarm schedule for BFS.

Fig. 6: Example schedules for statements s0 and s1 in Figure 2.

```
1  if (frontier.size < frontier.max_num_elems * 0.15) {
2     PUSH_edgeset<... // EdgeSetIterator with sched1
3  } else {
4     PULL_edgeset<... // EdgeSetIterator with sched2
5  }
```

Fig. 7: Host-side code with runtime condition generated based on the CompositeGPUSchedule in Figure 6a.

which inherits from the CompositeSchedule class shown in Table V. The CompositeGPUSchedule object is a hybrid schedule combining two AbstractSchedule objects (which could be other CompositeSchedule objects). The user also specifies the runtime criteria and its associated parameters. Here, the INPUT_SET_SIZE criteria is used with 0.15 as the criteria. This tells the compiler to generate code that chooses between sched1 and sched2, based on whether the input vertex set is above 15% of the total vertices in the graph. Figure 7 shows the generated code. The conditions and copies of the EdgeSetIterator with schedules sched1 and sched2 attached are created by the hardware-independent compiler and GraphVMs need not be aware of it. The compiler generates a nested if-then-else statement if multiple CompositeSchedule objects are combined.

IV. Evaluation

In this section, we demonstrate that UGC supports implementing optimizations that are critical for performance on the four architectures we target: CPUs, GPUs, Swarm, and the HammerBlade Manycore. We compare the performance of optimized code generated by the GraphVMs for each of the architectures with baseline, unoptimized code on 5 graph algorithms and up to 10 different graph inputs. Baseline code is generated by applying the default schedule for each GraphVM to the algorithm. For the optimized version, we tune the schedules for each application and graph pair, but always compile from exactly the same algorithm specification.

Cores	64 cores in 16 tiles (4 cores/tile), 3.5 GHz, x86-64 ISA, Haswell-like 4-wide OoO cores [35], 2 threads/core [4]
L1 Cache	32 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 Cache	1 MB, per-tile, 8-way, inclusive, 9-cycle latency
L3 Cache	64 MB, shared, static NUCA [48] (4 MB bank/tile), 16-way, inclusive, 12-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	Four 4×4 bidirectional meshes, 192-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns
Memory	8 controllers, 24 GB/s each, 120-cycle minimum latency
Queues	128 task queue entries/core (8192 total), 32 commit queue entries/core (2048 total)
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [17] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Commit	Tiles send updates to virtual time arbiter every 120 cycles

TABLE VI: Configuration of the 64-core Swarm system.

Cores	128 cores in 16×8 grid RISC-V 32-bit IMAF ISA 4KB Instruction Cache 4KB Data Scratchpad
Cache	128KB Total Capacity 32 Independent Banks 8-way Set Associative
NoC	Bidirectional 2D Mesh (32-bit data, 64-bit addr)
Memory	2 HBM2 channels 32 GB/s per channel 512 MB per channel

TABLE VII: HammerBlade Manycore configuration.

Graph	Vertex count	Edge count
RN [54]	1,971,281	5,533,214
RC [27]	14,081,816	33,866,826
RU [27]	23,947,347	57,708,624
PK [54]	1,632,803	30,622,564
HW [26]	1,139,905	112,751,422
LJ [54]	4,847,571	85,702,474
OK [73]	2,997,166	212,698,418
IC [26]	7,414,865	301,969,638
TW [73]	21,297,772	530,051,090
SW [73]	58,655,849	522,642,066

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

A. Methodology

CPU and GPU. We evaluate the GPU GraphVM on a system with an NVIDIA Tesla V100 GPU with 32 GB of GDDR5 main memory, 6 MB of L2 cache, and 128 KB of L1 cache per SM, with a total of 80 SMs. This is a Volta-generation GPU. We evaluate the CPU GraphVM on a dual-socket system with Intel Xeon E5-2695 v3 12-core CPUs, for a total of 24 cores and 48 hardware thread contexts. The machine has 128 GB of DDR3-1600 main memory and a 30 MB last-level cache per socket, and has Transparent Huge Pages (THP) enabled.

Swarm Simulation. We evaluate the Swarm GraphVM by running each algorithm's compiled code in full on the open-source Swarm architectural simulator [62, 91]. We model a 64-core Swarm CPU with parameters shown in Table VI, similar to prior work [4, 91]. We model wide out-of-order cores similar to the Haswell cores in the Xeon E5-2695 v3 used for the CPU GraphVM. We perform cycle-level simulation of Swarm with detailed core, network, and memory system models, and model task and speculation overheads in detail [4, 91].

HammerBlade Manycore Simulation. We model a HammerBlade Manycore system running at 1GHz with 16 columns and 8 rows of core tiles, with parameters shown in Table VII. We use detailed, cycle-accurate RTL simulation to model the RISC-V cores, network on chip, and LLC. The RTL for this manycore has been validated in silicon, and this configuration occupies approximately 3.5 mm^2 of die area. We model the HBM2 memory system with DRAMSim3 [56], a timing accurate simulator. Generated host code runs natively on an Intel Xeon Gold 6254 CPU, and host libraries interface directly with the simulator environment using SystemVerilog DPI.

Datasets. Table VIII lists the input graphs used in the evaluation, along with their sizes in vertices and edges. Out of the 10 graphs, Orkut (OK), Twitter (TW), LiveJournal (LJ), SinaWeibo (SW), Hollywood (HW), Pokec (PK), and Indochina (IC) have power-law degree distributions, while RoadUSA (RU), RoadNetCA (RN), and RoadCentral (RC) have bounded degree distributions. These datasets include social graphs, web graphs, and road graphs.

Algorithms. We evaluate all GraphVMs on five algorithms: PageRank, BFS, SSSP with Δ-stepping, connected components (CC) and betweenness centrality (BC). PageRank [66] and CC [8, 80] are topology-driven algorithms where all the edges are traversed in each iteration. These applications have massive parallelism each round. BC [9] and BFS are data-driven algorithms where only a set of active vertices are processed each round. SSSP with Δ-stepping is a priority-based algorithm where the vertices are processed in a priority order for greater work efficiency. UGC compiles a single source code specification for each algorithm, reusing the same application code for all different architectures. In real-world applications, these algorithms could be run many times on one graph or class of graph (e.g., one runs many iterations of PageRank, while BFS, BC, and SSSP may be rerun from different starting vertices), necessitating tuning the implementation to the characteristics of the graph and architecture for high efficiency.

Schedules. The performance of the GraphVMs heavily depends on the schedules specified. We manually wrote schedules to tune the implementation of each algorithm to the graph type (e.g., road graphs vs. social graphs). Schedule parameters were further tuned by sweeping the parameter space. Prior work [15, 93, 95] has also shown that techniques like autotuning can find high-performance schedules in relatively little time.

B. Performance of Optimized Code

Figure 8 shows the performance improvements produced by optimization passes in each of our four GraphVMs. The speedups reported here are over the baseline code generated by applying the default schedule. Both the baseline and optimized code are parallel, and all generated C++ is compiled with optimizations enabled in the backend compiler.

We now discuss how the hardware-specific optimizations in the GraphVMs produce these speedups.

C. CPU and GPU

The baseline schedule for CPUs and GPUs uses push-based traversal with vertex-based parallelism. UGC achieves large speedups (up to $53\times$) on both of the architectures on BFS and BC by using Hybrid (Push+Pull) traversals and tuning the input frontier representation. PageRank greatly benefits from EdgeBlocking and NUMA optimizations, which improve locality of random accesses by tiling for the last-level cache. SSSP on CPUs benefits from the bucket fusion optimization for road graphs. This is consistent with the speedups of the GraphIt compiler [93]. Finally, CC benefits from better load balancing techniques (ETWC) on GPUs, and from edge-aware

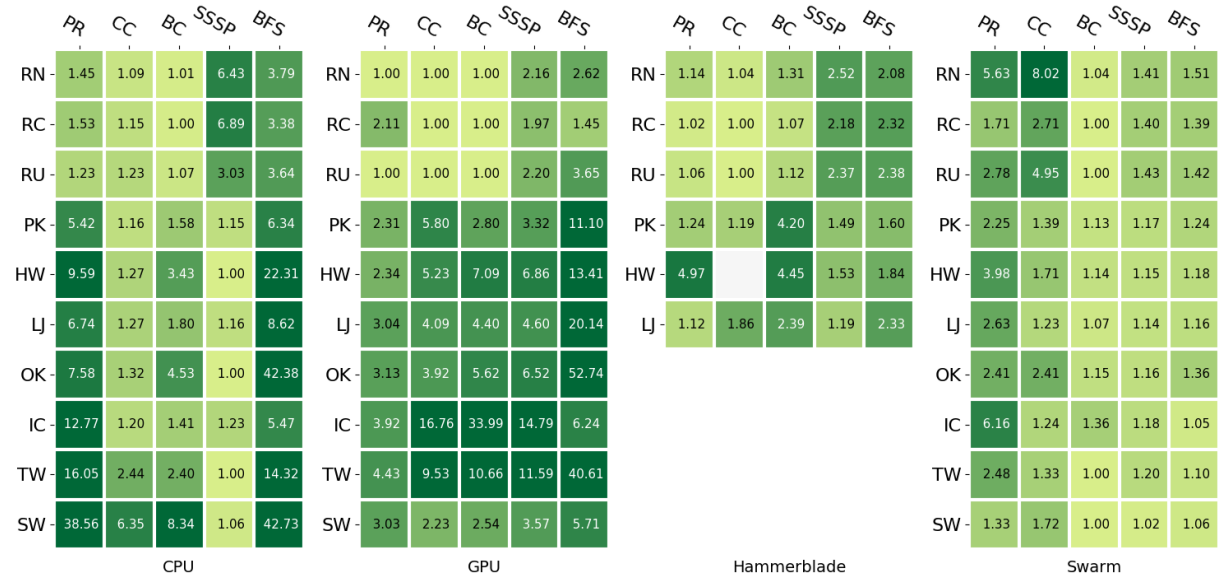


Fig. 8: Heatmap of speedups for the four evaluated architectures. Each cell reports the speedup of the optimized code over the baseline unoptimized version, with larger speedups in darker green. Columns correspond to algorithms, and rows correspond to graph inputs. Some graphs were not run on HammerBlade Manycore due to simulation time constraints.

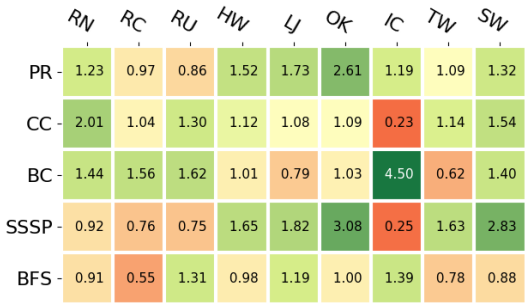
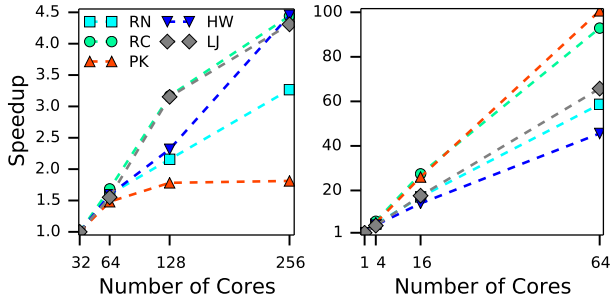


Fig. 9: Speedups of the GPU GraphVM over the next-best framework from Gunrock, GSwitch, or SEP-Graph.



(a) HammerBlade Manycore (b) Swarm

Fig. 10: Scalability of BFS on five graphs, shown across four sizes of HammerBlade Manycore and Swarm machines.

Graph	DRAM Stalls	Bandwidth	Speedup
LJ	0.78	3.03	1.19
HW	0.79	2.17	1.53
PC	0.83	3.02	1.49

TABLE IX: Impact of the HammerBlade blocked access optimization on SSSP. Reduction in DRAM stalls, improvement in memory bandwidth utilization, and overall speedup.

vertex-based parallelism on CPUs.

Figure 9 compares the performance of the GPU GraphVM with three state-of-the-art graph libraries that specifically target GPUs: Gunrock [87], GSwitch [60], and SEP-Graph [86]. These speedups are consistent with those of the GPU code generated from GraphIt [15]. UGC is consistently outperformed by SEP-Graph on SSSP when run on road graphs. SEP-Graph implements asynchronous execution to remove barriers between successive rounds of SSSP. UGC does not currently implement this optimization because it is very algorithm specific and cannot be generalized.

D. HammerBlade Manycore

Due to the costs of RTL simulation, we evaluate the HammerBlade Manycore GraphVM on 6 of the 10 input graphs and a subset of the total iterations for each application. For PR we simulate one iteration, and for the remaining applications, we simulate five representative iterations that cover a range of frontier densities and execution behavior. We use hybrid traversal in the baseline code of BFS, BC, and SSSP to decrease simulation times. The speedups reported in Figure 8 come from applying the HammerBlade Manycore-specific optimizations described in Section III-C4. BC, CC, and BFS benefit from alignment-based partitioning, while PR and SSSP use the blocking optimization due to their more compute-intensive nature. These optimizations better utilize the memory hierarchy and provide up to $4.97\times$ speedup over unoptimized code.

Figure 10a shows how performance scales on the HammerBlade Manycore. We ran our optimized BFS code on four different machine configurations: we hold the LLC capacity and number of columns (16) constant and vary the number of rows (2, 4, 8, and 16) to vary the total number of cores. The strong scaling indicates that the HammerBlade Manycore GraphVM can successfully exploit parallelism. We highlight BFS for this scaling study due to its high memory access to

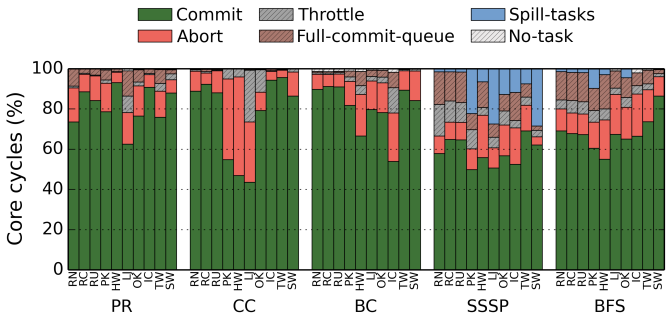


Fig. 11: Breakdown of how cores spend time for Swarm.

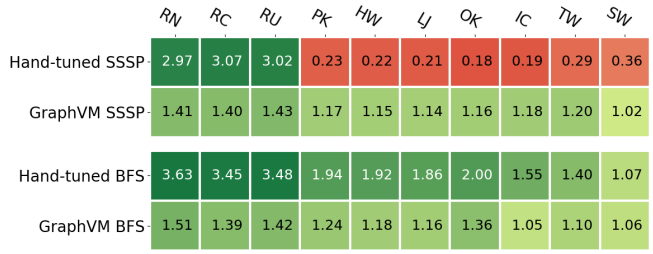


Fig. 12: Speedups of Swarm GraphVM optimized code and manually optimized assembly-level code from prior work, all relative to the Swarm GraphVM's default baseline code.

compute ratio.

Table IX demonstrates performance improvements for SSSP with Δ-stepping when the blocked-access optimization is applied on three selected input graphs. This optimization exploits memory parallelism to hide DRAM access latency in exchange for loading unused data and reducing effective bandwidth. For SSSP, we observe that this optimization decreases DRAM stalls, increases memory bandwidth utilization, and improves overall application performance.

E. Swarm

Figure 8 shows the speedup achieved by choosing an appropriate schedule for each algorithm and graph input, compared the Swarm GraphVM's default schedule. Swarm's T4 compiler [91] already applies many optimizations to uncover parallelism in serial code, and achieves good baseline performance in many cases. However, the Swarm GraphVM improves performance further by exploiting domain knowledge to choose optimizations.

On BFS and SSSP, converting `VertexSets` to tasks is responsible for the majority of the improvement on road graphs. This optimization avoids synchronization overheads between distance levels, by allowing tasks from different levels to execute speculatively in parallel. Additionally, all algorithms benefit from the Swarm GraphVM's diverse schedule options for task granularity and spatial hints. Fine-grained splitting with spatial hints allows trading increased task overheads for reduced cache line ping-ponging and abort costs. Finally, on CC and PageRank, some graphs featuring many high in-degree nodes benefit from a schedule that shuffles the order in which edges are processed, thus trading off locality to reduce aborts. This reordering is enabled by the Swarm GraphVM's domain knowledge that a valid result will still be produced if edges are visited in a different order within one round.

Table X compares the performance of optimized code generated by the CPU and Swarm GraphVMs. Since Swarm offers a superset of a CPU's features, the CPU code runs on the same Swarm hardware. On road graphs, the Swarm GraphVM consistently outperforms the CPU GraphVM using Swarm's speculative parallel execution of fine-grained tasks.

Graph	SSSP	BFS
RN	1.57	2.59
RC	2.04	2.56
RU	1.90	2.39

TABLE X: The Swarm GraphVM's speedup over the CPU GraphVM's best code on 64-core Swarm.

Figure 11 breaks down how cores spend time with the optimized schedules, averaged across the 64 cores: cores may spend time executing tasks that commit or abort; or idle due to the speculation-throttling heuristic [91], exhaustion of the commit queue, or lack of tasks to run. In some cases, cores also spend time spilling contents of overfull task queues to memory. We see that across all five algorithms, cores spend most of their time executing useful work that commits, demonstrating that the Swarm GraphVM exposes enough fine-grained parallelism through task spawns to utilize tens of cores. This is reflected in Swarm's strong scalability in Figure 10b. (Adding tiles to Swarm increases aggregate cache and queue capacity, sometimes yielding superlinear speedups.)

Prior work on Swarm has developed hand-tuned versions of BFS and SSSP [42, 43]. Figure 12 shows that the Swarm GraphVM versions are competitive with the manually tuned ones, especially on larger social graphs like TW and SW where the algorithms are memory-bound. The hand-tuned versions were tailored to work well on road graphs, which have low vertex degrees. As a result, the hand-tuned code for SSSP performs poorly on social graphs, where the Swarm GraphVM achieves much better performance by being selective in spawning tasks for the possibly many neighbors of each visited node. UGC makes it easy to bring a wide set of algorithms to developers of new graph processing architectures, and enables us to easily explore algorithm implementations that weren't obvious to the architecture's designers.

V. Related Work

There has been a large amount of work on both graph processing frameworks and on leveraging of common IRs to port applications to different architectures.

Common IRs for diverse architectures. Delite [18] introduces a new IR for parallel programs to target heterogenous architectures. However, Delite's IR is generic rather than specific to a particular domain. By customizing the IR specifically for the graph domain, UGC can perform optimizations that are otherwise infeasible in general C++ programs. Furthermore, Delite does not have an extensible scheduling language that allows users to specify optimizations for different targets. MLIR [53] is another proposed IR that is generic and not specific to a domain. Tensorflow [2] and TVM [22] have shown how a common IR can be used to apply machine learning optimizations across different architectures.

Graph processing frameworks. There has been a large body of work on graph processing for shared-memory [3, 31, 32, 39, 52, 69, 78, 79, 82, 83, 85, 92, 94], GPUs [12, 20, 24, 30, 36, 37, 38, 40, 46, 47, 49, 57, 58, 61, 63, 64, 67, 76, 80, 86, 87], and manycore architectures [21, 55, 69]. These frameworks support a limited set of optimizations, do not achieve consistently high performance across algorithms and graphs [15, 93, 95]and do not offer portability across architectures.

Abelian [31] uses the Galois framework as an interface for shared-memory CPU, distributed-memory CPU, and GPU platforms. However Abelian is not extensible enough to support new architectures. In contrast, UGC demonstrates state-of-the-art performance across different platforms.

Compilers for graph applications. IrGL [67] is a compiler framework that creates an intermediate representation specifically for graph applications on GPUs. IrGL introduces several optimizations for GPUs, but does not achieve state-of-the-art GPU performance [15, 86]. GraphIt [15, 93, 95] is a domain-specific language that expands the optimization space to outperform other CPU and GPU frameworks by decoupling algorithm from optimizations. UGC extends GraphIt by decoupling algorithms, optimizations, and hardware banckends to enable efficient implementations across different platforms.

VI. Conclusion

This paper has presented UGC, a novel graph processing framework that makes it easy to create compiler backends across diverse architectures. We introduced a new IR for graph processing, GraphIR, and showed how it can be used to implement GraphVMs for four different architectures. We demonstrated how UGC can reason about algorithmic and hardware-specific optimizations to generate high-performance code on all four architectures, and find that these optimizations can provide up to $53\times$ speedup over programmer-generated baseline implementations.

Acknowledgements

We thank Mark C. Jeffrey, Quan M. Nguyen, Hyun Ryong Lee and the anonymous reviewers for helpful discussions and feedback. This work was partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863, FA8650-18-2-7856; DARPA SDH under contract HR0011-18-3-0007; NSF grants SaTC-1563767, SaTC-1565446, SHF-1814969, and CAREER-1845763; DOE Early Career Award DE-SC0018947; a Sony research grant; and the DARPA/SRC JUMP ADA Center. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, "General-purpose graphics processor architectures," *Synthesis Lectures on Computer Architecture*, 2018.

[2] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI-12*, 2016.

[3] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "EmptyHeaded: A relational engine for graph processing," in *Proc. SIGMOD*, 2016.

[4] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez, "SAM: Optimizing multithreaded cores for speculative parallelism," in *Proc. PACT-26*, 2017.

[5] S. N. Agathos, A. Papadogiannakis, and V. V. Dimakopoulos, "Targeting the Parallella," in *Proc. Euro-Par*, 2015.

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ISCA*, 2015.

[7] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proc. PACT*, 2014.

[8] D. A. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Proc. ICPP*, 2005.

[9] D. A. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. ICPP*, 2006.

[10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. SC12*, 2012.

[11] S. Beamer, K. Asanović, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *Proc. IISWC*, 2015.

[12] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proc. PPoPP*, 2017.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. PPoPP*, 1995.

[14] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," in *Proc. SIGCOMM*, 2014.

[15] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, "Compiling graph applications for GPUs with GraphIt," in *Proc. CGO*, 2021.

[16] N. Bronson *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX ATC*, 2013.

[17] J. L. Carter and M. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. STOC-9*, 1977.

[18] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proc. PPoPP*, 2011.

[19] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*, 2001.

[20] S. Che, "GasCL: A vertex-centric graph model for GPUs," in *Proc. HPEC*, 2014.

[21] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal, "Efficient and simplified parallel graph processing over CPU and MIC," in *Proc. IPDPS*, 2015.

[22] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI-13*, 2018.

[23] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA a case study of breadth-first search," in *Proc. FPGA*, 2016.

[24] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. IPDPS*, 2014.

[25] S. Davidson *et al.*, "The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, 2018.

[26] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM TOMS*, 2011.

[27] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/.

[28] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proc. SPAA*, 2018.

[29] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," in *Proc. WWW*, 2018.

[30] A. Gaihre, Z. Wu, F. Yao, and H. Liu, "XBFS: eXploring runtime optimizations for breadth-first search on GPUs," in *Proc. HPDC*, 2019.

[31] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A compiler for graph analytics on distributed, heterogeneous platforms," in *Proc. Euro-Par*, 2018.

[32] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proc. PPoPP*, 2018.

[33] L. Gwennap, "Adapteva: More flops, less watts," *Microprocessor Report*, 2011.

[34] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. MICRO*, 2016.

[35] P. Hammarlund *et al.*, "Haswell: The fourth-generation Intel core processor," *IEEE Micro*, 2014.

[36] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *Proc. PACT-26*, 2017.

[37] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. HIPC*, 2007.

[38] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "MultiGraph: Efficient graph processing on GPUs," in *Proc. PACT-26*, 2017.

[39] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proc. ASPLOS-XVII*, 2012.

[40] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. PPoPP*, 2011.

[41] JEDEC, 2020. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[42] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-Centric execution of speculative parallel programs," in *Proc. MICRO*, 2016.

[43] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proc. MICRO-48*, 2015.

[44] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez, "Harmonizing speculative and non-speculative execution in architectures for ordered parallelism," in *Proc. MICRO-51*, 2018.

[45] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS," in *Proc. HPEC*, 2016.

[46] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-Efficient graph processing on GPUs," in *Proc. PACT-24*, 2015.

[47] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: vertex-centric graph processing on GPUs," in *Proc. HPDC*, 2014.

[48] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. ASPLOS-X*, 2002.

[49] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to GPUs," in *Proc. SIGMOD*, 2016.

[50] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," in *Proc. OOPSLA*, 2017.

[51] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *TOCS*, 2000.

[52] M. S. Lam, S. Guo, and J. Seo, "SociaLite: Datalog extensions for efficient social network analysis," in *Proc. ICDE*, 2013.

[53] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of Moore's law," 2020.

[54] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.

[55] D. Li, S. Chakradhar, and M. Becchi, "GRapid: A compilation and runtime framework for rapid prototyping of graph applications on many-core processors," in *Proc. ICPADS*, 2014.

[56] S. Li, R. S. Verdejo, P. Radojković, and B. Jacob, "Rethinking cycle accurate DRAM simulation," in *Proc. MEMSYS*, 2019.

[57] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proc. CC*, 2016.

[58] H. Liu and H. H. Huang, "SIMD-X: Programming and processing of graph algorithms on GPUs," in *Proc. USENIX ATC*, 2019.

[59] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, 2007.

[60] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on GPUs," in *Proc. PPoPP*, 2019.

[61] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. PPoPP*, 2012.

[62] MIT CSAIL, "The Swarm architecture." [Online]. Available: http://swarm.csail.mit.edu/

[63] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Proc. IPDPS*, 2013.

[64] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proc. ASPLOS-XXIII*, 2018.

[65] NVIDIA, "CUDA C++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2019.

[66] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford, Technical Report, 1999.

[67] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proc. OOPSLA*, 2016.

[68] S. Pallottino and M. G. Scutellà, *Shortest Path Algorithms In Transportation Models: Classical and Innovative Aspects*, 1998.

[69] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren, "GraphPhi: efficient parallel graph processing on emerging throughput-oriented architectures," in *Proc. PACT-26*, 2018.

[70] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. PLDI*, 2013.

[71] C. Ramey, "TILE-Gx100 ManyCore processor: Acceleration interfaces and architecture," in *Proc. HotChips*, 2011.

[72] D. Richmond, A. Althoff, and R. Kastner, "Synthesizable higher-order functions for C++," *IEEE TCAD*, 2018.

[73] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. AAAI-29*, 2015.

[74] A. Shajii, I. Numanagić, R. Baghdadi, B. Berger, and S. Amarasinghe, "Seq: a high-performance language for bioinformatics," in *Proc. OOPSLA*, 2019.

[75] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "GraphJet: Real-time content recommendations at Twitter," *VLDB Endow.*, 2016.

[76] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin, "Frog: Asynchronous graph processing on GPU with hybrid coloring model," *TKDE*, 2017.

[77] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on GPUs: A survey," *ACM Computing Surveys (CSUR)*, 2018.

[78] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.

[79] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *Proc. DCC*, 2015.

[80] J. Soman, K. Kishore, and P. J. Narayanan, "A fast GPU algorithm for graph connectivity," in *Proc. IPDPSW*, 2010.

[81] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *Proc. ISCA-44*, 2017.

[82] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proc. ICS*, 2017.

[83] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *VLDB Endow.*, 2015.

[84] M. B. Taylor *et al.*, "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," in *Proc. ISCA-31*, 2004.

[85] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proc. ASPLOS-XXII*, 2017.

[86] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "SEP-Graph: finding shortest execution paths for graph processing under a hybrid framework on GPU," in *Proc. PPoPP*, 2019.

[87] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. PPoPP*, 2016.

[88] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *Proc. IISWC*, 2014.

[89] C. Yang, A. Buluc, and J. D. Owens, "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU," *arXiv preprint arXiv:1908.01407*, 2019.

[90] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. KDD*, 2018.

[91] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *Proc. ISCA-47*, 2020.

[92] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. PPoPP*, 2015.

[93] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with GraphIt," in *Proc. CGO*, 2020.

[94] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. BigData*, 2017.

[95] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," in *Proc. OOPSLA*, 2018.