GraphIt: A High-Performance Graph DSL

YUNMING ZHANG, MIT CSAIL, USA
MENGJIAO YANG, MIT CSAIL, USA
RIYADH BAGHDADI, MIT CSAIL, USA
SHOAIB KAMIL, Adobe Research, USA
JULIAN SHUN, MIT CSAIL, USA
SAMAN AMARASINGHE, MIT CSAIL, USA

The performance bottlenecks of graph applications depend not only on the algorithm and the underlying hardware, but also on the size and structure of the input graph. As a result, programmers must try different combinations of a large set of techniques, which make tradeoffs among locality, work-efficiency, and parallelism, to develop the best implementation for a specific algorithm and type of graph. Existing graph frameworks and domain specific languages (DSLs) lack flexibility, supporting only a limited set of optimizations.

This paper introduces **GraphIt**, a new DSL for graph computations that generates fast implementations for algorithms with different performance characteristics running on graphs with different sizes and structures. GraphIt separates what is computed (algorithm) from how it is computed (schedule). Programmers specify the algorithm using an *algorithm language*, and performance optimizations are specified using a separate *scheduling language*. The algorithm language simplifies expressing the algorithms, while exposing opportunities for optimizations. We formulate graph optimizations, including edge traversal direction, data layout, parallelization, cache, NUMA, and kernel fusion optimizations, as tradeoffs among locality, parallelism, and work-efficiency. The scheduling language enables programmers to easily search through this complicated tradeoff space by composing together a large set of edge traversal, vertex data layout, and program structure optimizations. The separation of algorithm and schedule also enables us to build an autotuner on top of GraphIt to automatically find high-performance schedules. The compiler uses a new scheduling representation, the *graph iteration space*, to model, compose, and ensure the validity of the large number of optimizations. We evaluate GraphIt's performance with seven algorithms on graphs with different structures and sizes. GraphIt outperforms the next fastest of six state-of-the-art shared-memory frameworks (Ligra, Green-Marl, GraphMat, Galois, Gemini, and Grazelle) on 24 out of 32 experiments by up to 4.8×, and is never more than 43% slower than the fastest framework on the other experiments. GraphIt also reduces the lines of code by up to an order of magnitude compared to the next fastest framework.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Software and its engineering → Parallel programming languages**; **Domain specific languages**;

Additional Key Words and Phrases: Compiler Optimizations, Code Generation, Big Data

ACM Reference Format:
Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (November 2018), 30 pages. https://doi.org/10.1145/3276491

Authors' addresses: Yunming Zhang, MIT CSAIL, USA, yunming@mit.edu; Mengjiao Yang, MIT CSAIL, USA, mengjiao@mit.edu; Riyadh Baghdadi, MIT CSAIL, USA, baghdadi@mit.edu; Shoaib Kamil, Adobe Research, USA, kamil@adobe.com; Julian Shun, MIT CSAIL, USA, jshun@mit.edu; Saman Amarasinghe, MIT CSAIL, USA, saman@csail.mit.edu.

2475-1421/2018/11-ART121
https://doi.org/10.1145/3276491

1 INTRODUCTION

In recent years, large graphs with billions of vertices and trillions of edges have emerged in many domains, such as social network analytics, machine learning, and biology. Extracting information from these graphs often involves running algorithms for identifying important vertices, finding connections among vertices, and detecting communities of interest. Speeding up these algorithms can enhance the efficiency of data analytics applications and improve the quality of web services [Eksombatchai et al. 2018; Fu et al. 2017; Sharma et al. 2016].

It is difficult to implement high-performance graph algorithms. The performance bottlenecks of these algorithms depend not only on the algorithm and the underlying hardware, but also on the size and structure of the graph [Beamer et al. 2015]. As a result, different algorithms running on the same machine, or even the same algorithm running with different types of graphs on the same machine, can exhibit different performance bottlenecks. For example, some algorithms, such as PageRank, spend most of the time working on the entire graph, while traversal algorithms, such as Breadth-First Search (BFS), work on a smaller subgraph at a time. In addition, running the same algorithm on a social network with small diameter and power-law degree distribution exposes different performance bottlenecks compared to running on a road network with large diameter and uniform degree distribution.

Graph optimizations make tradeoffs among locality, parallelism and work-efficiency [Beamer et al. 2012, 2017; Kiriansky et al. 2016; Shun and Blelloch 2013; Zhang et al. 2017] to improve performance. Since many graph algorithms have very different performance bottlenecks, and optimizations make different tradeoffs, one optimization can significantly boost the performance for some graph algorithms on certain types of graphs, while hurting the performance of other algorithms or the same algorithm running on other types of graphs [Malicevic et al. 2017]. Programmers must iterate over multiple implementations of the same algorithm to identify the best combination of optimizations for a specific algorithm and input data.

Existing graph frameworks perform well for a subset of algorithms for specific types of input, but have suboptimal performance on algorithms with different bottlenecks and graphs of different sizes and structures [Beamer et al. 2015; Satish et al. 2014]. Developers writing performance-critical applications cannot afford such a level of slowdown. This performance inconsistency exists because each framework was designed to support only a limited set of optimization techniques, and does not allow for easy exploration of the large space of optimizations. It is infeasible to write hand-optimized code for every combination of algorithm and input type. A compiler approach that generates efficient implementations from high-level specifications is therefore a good fit. However, existing graph domain specific languages (DSLs) [Aberger et al. 2016; Hong et al. 2012; Lam et al. 2013] do not support composition of optimizations or expose comprehensive performance tuning capabilities to programmers.

We introduce GraphIt,[1] a new graph DSL that produces efficient implementations with performance competitive with or faster than state-of-the-art frameworks for a diverse set of algorithms running on graphs with different sizes and structures. GraphIt achieves good performance by enabling programmers to easily find the best combination of optimizations for their specific algorithm and input graph. In this paper, we focus on providing mechanisms that make it possible to easily and productively explore the space of optimizations.

GraphIt separates algorithm specifications from the choice of performance optimizations. Programmers specify the algorithm using an *algorithm language* based on high-level operators on sets of vertices and edges. Performance optimizations are specified using a separate *scheduling language*. The algorithm language simplifies expressing algorithms and exposes opportunities for

[1]The GraphIt compiler is available under the MIT license at http://graphit-lang.org/

optimizations by separating edge processing logic from edge traversal, edge filtering, vertex deduplication, and synchronization logic. We formulate graph optimizations, including edge traversal direction, data layout, parallelization, cache, NUMA, and kernel fusion optimizations, as tradeoffs among locality, parallelism and work-efficiency. The scheduling language enables programmers to easily search through the complicated tradeoff space by composing a large set of edge traversal, vertex data layout, and program structure optimizations.

GraphIt introduces novel scheduling representations for edge traversal, vertex data layout, and program structure optimizations. Inspired by iteration space theory for dense loops [Padua and Wolfe 1986; Wolf and Lam 1991], we introduce an abstract *graph iteration space* model to represent, compose, and ensure the validity of edge traversal optimizations. We encode the graph iteration space in the compiler's intermediate representation to guide program analyses and code generation.

The separation of algorithm and schedule also enables GraphIt to search for high-performance schedules automatically. The large scheduling space and long running time of the applications make it costly to do an exhaustive search. We show that it is possible to discover schedules with good performance in much less time using autotuning. Programmers familiar with graph optimizations can leverage their expertise to tune performance using the scheduling language directly.

We perform a comprehensive analysis of GraphIt's performance using six state-of-the-art shared-memory frameworks (Ligra [Shun and Blelloch 2013], GraphMat [Sundaram et al. 2015], Galois [Nguyen et al. 2013], Green-Marl [Hong et al. 2012], Gemini [Zhu et al. 2016], and Grazelle [Grossman et al. 2018]) with seven algorithms consisting of PageRank (PR), Breadth-First Search (BFS), Connected Components (CC) with synchronous label propagation, Single Source Shortest Paths (SSSP) with frontier-based Bellman-Ford, Collaborative Filtering (CF), Betweenness Centrality (BC), and PageRankDelta (PRDelta), running on real-world graphs with different sizes and structures. Our experiments show that GraphIt outperforms the next fastest of the shared-memory frameworks on 24 out of 32 experiments by up to 4.8×, and is never more than 43% slower than the fastest framework on the other experiments. For each framework and DSL, we show a heat map of slowdowns compared to the fastest of all seven frameworks and DSLs in Fig. 1. GraphIt does not introduce any new optimizations. Instead, the DSL achieves competitive or better performance compared to other frameworks by generating efficient implementations of known combinations of optimizations, and finding previously unexplored combinations by searching through a much larger space of optimizations. GraphIt also reduces the lines of code compared to the next fastest framework by up to an order of magnitude.

This paper makes the following contributions:

- A systematic analysis of the fundamental tradeoffs among locality, work-efficiency, and parallelism in graph optimizations (Section 2).
- A high-level algorithm language that separates edge processing logic from edge traversal, synchronization, updated vertex tracking, and deduplication logic (Section 3).
- A new scheduling language that allows programmers to explore the tradeoff space by composing edge traversal, vertex data layout, and program structure optimizations (Section 4).
- A novel scheduling representation, the graph iteration space model, that can represent, combine and reason about the validity of various edge traversal optimizations (Section 5).
- A compiler that leverages program analyses on the algorithm language and an intermediate representation that encodes the graph iteration space to generate efficient and valid implementations for different combinations of optimizations. (Section 6).
- A comprehensive evaluation of GraphIt that shows it is faster than the next fastest state-of-the-art framework on 24 out of 32 experiments by up to 4.8×, and never more than 43% slower than the fastest framework on the other experiments (Section 7).

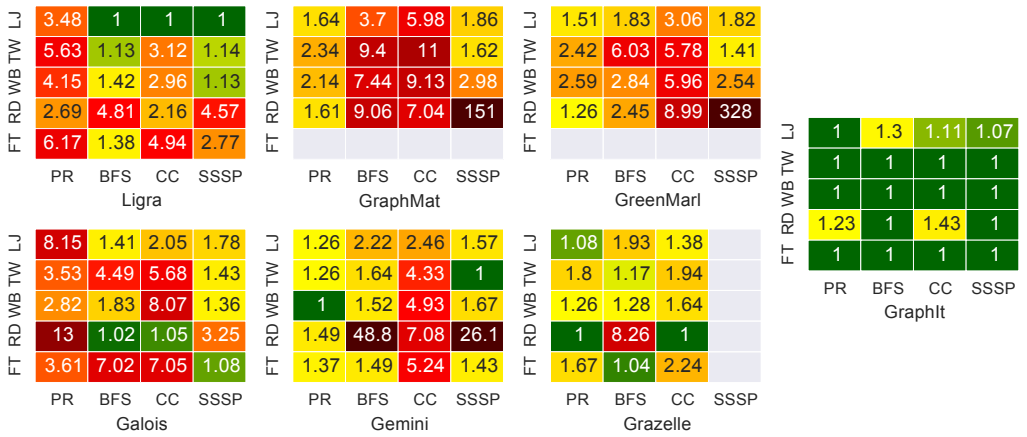

Fig. 1. A heat map of slowdowns of various frameworks compared to the fastest of all frameworks for PageRank (PR), Breadth-First Search (BFS), Connected Components (CC) using label propagation, and Single Source Shortest Paths (SSSP) using Bellman-Ford, on five graphs with varying sizes and structures (LiveJournal (LJ), Twitter (TW), WebGraph (WB), USAroad (RD), and Friendster (FT)). Lower numbers (green) are better, with one being the fastest for the specific algorithm running on the specific graph. Gray means that either an algorithm or a graph is not supported by the framework. We try to use the same algorithms across different frameworks. For Galois, we used the asynchronous algorithm for BFS, and the Ligra algorithm for SSSP.

2 TRADEOFF SPACE AND OPTIMIZATIONS

While the effects of various optimizations are well-known to expert programmers, we believe that we are the first to characterize the optimization tradeoff space for graph optimizations. Our tradeoff space includes three properties of graph programs—locality, work-efficiency, and parallelism. **Locality** refers to the amount of spatial and temporal reuse in a program. Increasing the amount of reuse improves locality due to increasing the number of cache hits. In a NUMA system, locality also refers to the memory location relative to the processor. Increasing the ratio of local memory accesses to remote memory accesses improves locality. **Work-efficiency** is the inverse of the number of instructions, where each instruction is weighted according to the number of cycles that it takes to execute assuming a cold cache. Reducing the number of instructions improves work-efficiency. **Parallelism** refers to the relative amount of work that can be executed independently by different processing units, which is often affected by the load balance and synchronization among processing units. Increasing parallelism improves performance by taking advantage of more processing units, and helping to hide the latency of DRAM requests. We use PageRankDelta (described in Section 2.1 and shown in Fig. 2) as an example to illustrate the effects of various optimizations on these three metrics. Each optimization can affect multiple properties in

```
1  Rank = {0, . . . , 0}                                   ▷ Length V array
2  DeltaSum = {0.0, . . . , 0.0}                           ▷ Length V array
3  Delta = {1/V, . . . , 1/V}                              ▷ Length V array
4  procedure PageRankDelta(Graph G, α, ϵ)
5      Frontier = { G.vertices }
6      for round ∈ {1, . . . ,MaxIter} do
7          NextFrontier = {}
8          parallel for src : Frontier do
9              for dst : G.getOutNgh[src] do
10                 AtomicAdd(DeltaSum[dst], Delta[src]/G.OutDegree[src])
11         parallel for v : G.vertices do
12             if round == 1 then
13                 BaseScore = (1.0 − α)/V
14                 Delta[v] = α · (DeltaSum[v])+ BaseScore
15                 Delta[v] −= 1/V
16             else
17                 Delta[v] = α · (DeltaSum[v])
18             Rank[v] += Delta[v]
19             DeltaSum[v] = 0
20             if |Delta[v]|> ϵ·Rank[v] then
21                 NextFrontier.add(v)
22         Frontier = NextFrontier
```

Fig. 2. PageRankDelta (SparsePush).

the tradeoff space, either positively or negatively. The complex tradeoff space motivates the design of GraphIt's scheduling language and compiler, which can be used to easily search for points in the tradeoff space that achieve high performance.

2.1 PageRankDelta

PageRankDelta [Shun and Blelloch 2013] is a variant of the standard PageRank algorithm [Page et al. 1999] that computes the importance of vertices in a graph. It maintains an array of ranks, and on each iteration, updates the ranks of all vertices based on the ranks of their neighbors weighted by their neighbors' out-degrees. PageRankDelta speeds up the computation by updating only the ranks of vertices whose ranks have changed significantly from the previous iteration.

The pseudocode for PageRankDelta is shown in Fig. 2, where V is the number of vertices in the graph, $0 \leq \alpha \leq 1$ is the damping factor that determines how heavily to weight the neighbors' ranks during the update, and $\epsilon \geq 0$ is a constant that determines whether a vertex's rank has changed sufficiently. For simplicity, we update the ranks of vertices for MaxIter number of iterations, although the code can easily be modified to terminate based on a convergence criterion. The algorithm maintains the set of vertices whose ranks (stored in the Rank array) have changed significantly from the previous iteration in the variable Frontier (represented as a sparse array). We will refer to this as the *active set* of vertices, or the *frontier*. Initially all vertices are in the active set (Line 5). On each iteration, each vertex in the active set sends its Delta (change in Rank value) from the previous iteration to its out-neighbors by incrementing the DeltaSum values of its neighbors (Lines 8–10). Since vertices are processed in parallel, the updates to DeltaSum must be atomic. Then in parallel, all vertices compute their own Delta and Rank values based on their DeltaSum value and α (Lines 11–18). Delta is computed differently for the first iteration. If the Delta of the vertex is larger than ϵ times its Rank, then the vertex is active for the next iteration and is added to the next frontier (Lines 20–21).

2.2 Graph Optimizations

We describe the tradeoffs of the optimizations listed in Table 1 with PageRankDelta (Table 11 describes which optimizations are supported by which frameworks and DSLs). Table 1 contains the effect of optimizations relative to the baseline in Fig. 2, which we refer to as *SparsePush*.

Direction Optimization and Frontier Data Structure. Lines 8–10 of SparsePush iterate over the outgoing neighbors of each vertex, and update the DeltaSum value of the neighbor. *DensePull* (Fig. 3 (left)) is a different traversal mode where each vertex iterates over its incoming neighbors that are in the active set, and updates its own DeltaSum value. DensePull increases parallelism relative to SparsePush since it loops over all vertices in the graph. This increases work compared to SparsePush, which only loops over vertices in the active set. The update to the DeltaSum array no longer requires atomics since an entry will

Table 1. Effect of optimizations on the different properties of the tradeoff space relative to the baseline SparsePush. For each property, ↑ means positive impact on performance, ↓ means negative impact on performance, ↕ means it could increase or decrease depending on various factors (described in the text), and no entry means no effect on performance.

Optimization	Locality	Work-efficiency	Parallelism
DensePull		↕	↕
DensePush		↕	↑
DensePull-SparsePush		↕	↕
DensePush-SparsePush		↕	↑
edge-aware-vertex-parallel		↓	↑
edge-parallel		↓	↑
bitvector	↑	↓	
vertex data layout	↕	↓	
cache partitioning	↑	↓	
NUMA partitioning	↑	↓	↓
kernel fusion	↑	↓	

```
1 parallel for dst : G.vertices do
2     for src : G.getInNgh[dst] do
3         if src ∈ Frontier then
4             DeltaSum[dst] += Delta[src] / G.OutDegree[src]
```

```
1 parallel for src : G.vertices do
2     if src ∈ Frontier then
3         for dst : G.getOutNgh[src] do
4             AtomicAdd(DeltaSum[dst],Delta[src] / G.OutDegree[src])
```

Fig. 3. DensePull (left) and DensePush (right).

not be updated in parallel, and this reduces synchronization overhead. Instead of doing random writes as in SparsePush, DensePull does random reads and mostly sequential writes, which are cheaper. For some algorithms (e.g., breadth-first search), the inner loop over the in-neighbors in DensePull can exit early to reduce overall work. Therefore, the overall number of edges traversed could increase or decrease. A detailed performance study of the two traversal methods can be found in Beamer et al. [2012] and Besta et al. [2017]. We can further use *bitvectors* instead of boolean arrays to keep track of vertices on the frontier for the DensePull direction. A dense frontier implemented using a bitvector improves spatial locality, but requires extra work to compress the boolean array.

DensePush (Fig. 3 (right)) loops through all vertices and checks if each one is on the frontier instead of only looping over frontier vertices as in SparsePush. While iterating over all vertices reduces work-efficiency, this could be offset by not having to maintain the frontier in a sparse format. Parallelism increases as there is more parallel work when looping over all vertices.

Hybrid traversal modes (*DensePull-SparsePush* and *DensePush-SparsePush*) use different directions (SparsePush, DensePull, and DensePush) in different iterations based on the size of the active set to improve work-efficiency [Beamer et al. 2012; Shun and Blelloch 2013]. In PageRankDelta, the number of vertices in the frontier gradually decreases as the ranks of vertices converge. In the early iterations, DensePull is preferred due to lower synchronization overheads and avoiding random writes. As the frontier shrinks, SparsePush is preferred due to the fewer number of vertices that need to be traversed. *DensePull-SparsePush* computes the sum of out-degrees of the frontier vertices and uses DensePull if the sum is above some threshold, and SparsePush otherwise. However, computing the sum of out-degrees of the vertices in the active set in every iteration incurs significant overhead if one direction is always better than the other.

GraphIt is able to support all of these traversal directions whereas existing frameworks only support a subset of them. GraphIt also supports both bitvectors and boolean arrays for the frontier representation in dense traversals, as well as the sparse array representation for sparse traversals.

Parallelization. For each traversal mode, there are different methods for parallelization. The parallelization shown in Fig. 2 and 3 processes the vertices in parallel. This approach (*vertex-parallel*) works well on algorithms and inputs where the workload of each vertex is similar. However, if the degree distribution is skewed and the workload of each vertex is proportional to the number of incident edges, this approach can lead to significant load imbalance. For these workloads, an edge-aware vertex-parallel scheme (*edge-aware-vertex-parallel*) can be more effective. This approach breaks up the vertices into a number of vertex chunks, where each chunk has approximately the same number of edges. However, this scheme reduces work-efficiency due to having to compute the sum of degrees of vertices in each chunk. Finally, we can parallelize across all edges, instead of just vertices, by parallelizing the inner loop of the edge-traversal code computing DeltaSum in Fig. 2 and 3. This method (*edge-parallel*) improves parallelism but reduces work-efficiency due to the overhead of work-stealing in the inner loop and atomic updates needed for synchronization. For graphs with a regular degree distribution, using static parallelism instead of work-stealing parallelism can sometimes reduce runtime overhead and improve performance. GraphIt supports all three modes of parallelism while existing frameworks only support one or two.

Cache Partitioning. Cache partitioning tries to keep random accesses within the last level cache (LLC) to improve locality. This optimization first partitions the vertices into p segments

$(V_0, V_1, \ldots, V_{p-1})$, which correspond to the range of source vertexsets in the pull mode or the destination vertexsets in the push mode for each Segmented Subgraph (SSG). For the pull mode, incoming edges (src, dst) are assigned to SSG_i if $src \in V_i$, and sorted by dst. For the push mode, outgoing edges (src, dst) are assigned to SSG_i if $dst \in V_i$, and sorted by src. Each SSG is processed before moving on to the next. V_i controls the range of random memory accesses through segmenting the original graph. If we fit V_i into LLC, we can significantly reduce the number of random DRAM accesses when processing each SSG. Cache partitioning improves locality but sacrifices work-efficiency due to vertex data replication from graph partitioning and merging partial results [Beamer et al. 2017; Nishtala et al. 2007; Zhang et al. 2017]. Fine-tuning the number of SSGs can reduce this overhead. Most existing frameworks do not support cache partitioning.

NUMA Optimizations. NUMA partitioning improves locality by minimizing slow inter-socket memory accesses [Sun et al. 2017; Zhang et al. 2015; Zhu et al. 2016]. This optimization partitions the graph into a set of Segmented Subgraphs (SSGs) in the same way as cache partitioning in order to limit the range of random memory access. While the cache partitioning optimization processes one SSG at a time across all sockets, NUMA partitioning executes multiple SSGs in parallel on different sockets. Each SSG and the threads responsible for processing the subgraph are bound to the same NUMA socket. The intermediate results collected on each socket are merged at the end of each iteration. As with cache partitioning, NUMA partitioning improves locality but reduces work-efficiency due to vertex data replication from graph partitioning and the additional merge phase. Parallelism might also decrease for highly skewed graphs due to workload imbalance among SSGs [Sun et al. 2017]. For algorithms with performance bottlenecked on load imbalance instead of inter-socket memory accesses, simply using an interleaved allocation across sockets can result in better performance. GraphIt and a subset of existing frameworks support NUMA optimizations.

Vertex Data Layout Optimizations. The layout of vertex data can significantly affect the locality of memory accesses. Random accesses to the same index of two separate arrays (e.g., the Delta and OutDegree arrays in PageRankDelta) can be changed into a single random access to an array of structs to improve spatial locality. However, grouping together fields that are not always accessed together expands the working set and hurts the locality of the data structures. Vertex data layout optimizations reduce work-efficiency due to the extra overhead for reorganizing the data. GraphIt supports both arrays of structs and structs of arrays.

Program Structure Optimizations. When two graph kernels have the same traversal pattern (they process the same vertices/edges on each iteration), we can fuse together the edge traversals and transform their data structures into an array of structs. We refer to this optimization as *kernel fusion*. This improves spatial locality by enabling the program to access the fields of the two kernels together when traversing the edges. Additional work is incurred for performing the AoS-SoA optimization, but this is usually small compared to the rest of the algorithm.

3 ALGORITHM LANGUAGE

GraphIt leverages an algorithm and a scheduling language to compose graph optimizations, enabling programmers to easily navigate the complex performance tradeoff space described in Section 2. The algorithm language can express a variety of algorithms, while exposing opportunities for optimizations. We use PageRankDelta (GraphIt code shown in Fig. 4) to showcase the language.

3.1 Data Model

GraphIt's data model consists of elements, vertexsets and edgesets, and vertex and edge data. The programmer first defines vertex and edge element types (Vertex and Edge on Lines 1–2 of Fig. 4). GraphIt supports multiple types of user-defined vertices and edges, which is important

```
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 damp : double = 0.85;
6 const base_score : double = (1.0 - damp)/vertices.size();
7 const epsilon : double = 0.1;
8 const OutDegree : vector{Vertex}(int) = edges.getOutDegrees();
9 Rank : vector{Vertex}(double) = 0;
10 DeltaSum : vector{Vertex}(double) = 0.0;
11 Delta : vector{Vertex}(double) = 1.0/vertices.size();
12 func updateEdge(src : Vertex, dst : Vertex)
13     DeltaSum[dst] += Delta[src]/OutDegree[src];
14 end
15 func updateVertexFirstRound(v : Vertex) -> output : bool
16     Delta[v] = damp * (DeltaSum[v]) + base_score;
17     Rank[v] += Delta[v];
18     Delta[v] = Delta[v] - 1.0/vertices.size();
19     output = fabs(Delta[v] > epsilon*Rank[v]);
20     DeltaSum[v] = 0;
21 end
22 func updateVertex(v : Vertex) -> output : bool
23    Delta[v] = DeltaSum[v] * damp;
24    Rank[v] += Delta[v];
25    DeltaSum[v] = 0;
26    output = fabs(Delta[v]) > epsilon * Rank[v];
27 end
28 func main()
29     var V : int = vertices.size();
30     var Frontier : vertexset{Vertex} = new vertexset{Vertex}(V);
31     for i in 1:maxIters
32         #s1# edges.from(frontier).apply(updateEdge);
33         if i == 1
34             Frontier = vertices.filter(updateVertexFirstRound);
35         else
36             Frontier = vertices.filter(updateVertex);
37         end
38     end
39 end
```

Fig. 4. GraphIt code for PageRankDelta.

for algorithms that work on multiple graphs. After defining element types, the programmer can construct vertexsets and edgesets. Lines 3–4 of Fig. 4 show the definitions of an edgeset, *edges*, and vertexset, *vertices*. Each element of the edgeset is of Edge type (specified between "{}"), and the source and destination of the edge is of Vertex type (specified between "( )"). The edgeset declaration supports edges with different types of source and destination vertices (e.g., in a bipartite graph). *vertices* uses the `getVertices` method on the edgeset to obtain the union of source and destination vertices of *edges*. Data for vertices and edges are defined as vectors associated with an element type denoted using the {} syntax (Lines 8–11).

3.2 Language Constructs and Operators

The language constructs of GraphIt (shown in Table 2) separate edge processing logic from edge traversal, edge filtering (`from`, `to`, `srcFilter`, and `dstFilter`), atomic synchronization, and modified vertex deduplication and tracking logic (`apply` and `applyModified`). This separation enables the compiler to represent the algorithm at a high level, exposing opportunities for edge traversal and vertex data layout optimizations. Moreover, it frees the programmer from specifying low-level implementation details, such as synchronization and deduplication logic.

In the GraphIt code for PageRankDelta (Fig. 4), the `from` operator (Line 32) ensures that only edges whose source vertex is in the frontier are traversed, and the `apply` operator uses the `updateEdge` function on the selected edges to compute DeltaSum, corresponding to Lines 8–10 of Algorithm 2. This separation enables the compiler to generate complex code for different traversal modes

Table 2. Vertexset and Edgeset API. `disable_deduplication` is an optional parameter.

Set Operators	Return Type	Description
`size()`	int	Returns the size of the set.
Vertexset operators		
`filter(func f)`	vertexset	Filters out vertices where f(vertex) returns true.
`apply(func f)`	none	Applies f(vertex) to every vertex.
Edgeset operators		
`from(vertexset vset)`	edgeset	Filters out edges whose source vertex is in the input vertexset.
`to(vertexset vset)`	edgeset	Filters out edges whose destination vertex is in the input vertexset.
`filter(func f)`	edgeset	Filters out edges where f(edge) returns true.
`srcFilter(func f)`	edgeset	Filters out edges where f(source) returns true.
`dstFilter(func f)`	edgeset	Filters out edges where f(destination) returns true.
`apply(func f)`	none	Applies f(source, destination) to every edge.
`applyModified(func f, vector vec, [bool disable_deduplication])`	vertexset	Applies f(source, destination) to every edge. Returns a vertexset that contains destination vertices whose entry in the vector vec has been modified in `f`. The programmer can optionally disable deduplication within modified vertices. Deduplication is enabled by default.

and parallelization optimizations, while inserting appropriate data access and synchronization instructions for the `updateEdge` function. **`#s1#`** is a label used in the scheduling language (explained in Section 4). Lines 34 and 36 of Fig. 4 compute the updated Delta and Rank values by applying updateVertexFirstRound and updateVertex functions on every vertex. Vertices with Delta greater than epsilon of their Rank are returned as the next frontier, corresponding to Lines 11–21 of Algorithm 2. As shown in Table 2, GraphIt provides various operators on vertexsets and edgesets to express graph algorithms with different traversal and update logic. The `applyModified` operator tracks which vertices have been updated during the edge traversal and outputs a vertexset containing just those vertices. By default, `applyModified` ensures that each vertex is added only once to the output vertexset. However, the programmer can optionally disable deduplication for algorithms that are guaranteed to insert each vertex only once (e.g., BFS) for better performance.

We demonstrate how GraphIt simplifies the expression of the algorithm by showing Ligra's implementation of the edge update function in Fig. 5 (note that the 16 lines of Ligra code shown correspond to only 3 lines in GraphIt's implementation in Fig. 4). Ligra requires the programmer to specify edge processing (Lines 8–9, 12–14), edge filtering (Line 16), deduplication and modification tracking (Lines 10 and 15), and synchronization logic (Lines 12–14). GraphIt only requires the programmer to specify the edge processing logic in this case.

```
1  template <class vertex>
2  struct PR_F {
3    vertex* V;
4    double* Delta, *nghSum;
5    PR_F(vertex* _V, double* _Delta, double* _nghSum) :
6      V(_V), Delta(_Delta), nghSum(_nghSum) {}
7    inline bool update(uintE s, uintE d){
8      double oldVal = nghSum[d];
9      nghSum[d] += Delta[s]/V[s].getOutDegree();
10     return oldVal == 0;}
11   inline bool updateAtomic (uintE s, uintE d) {
12     volatile double oldV, newV;
13     do { oldV = nghSum[d]; newV = oldV + Delta[s]/V[s].getOutDegree();
14     } while(!CAS(&nghSum[d],oldV,newV));
15     return oldV == 0.0;}
16   inline bool cond (uintE d) { return cond_true(d); }};
```

Fig. 5. Ligra's PageRankDelta edge update function, corresponding to Lines 12–14 of Fig. 4 in GraphIt's PageRankDelta example.

GraphIt also supports traditional control flow constructs such as `for`, `while`, and `if` for expressing fixed iteration loops, loops until convergence, and conditional control flow. After setting up a new vertexset called Frontier, Line 31 in Fig. 4 uses a `for` loop to iterate maxIters times. An alternative implementation could use a while loop that iterates until the ranks of all vertices stabilize.

Table 3. GraphIt Scheduling Language functions. The default option for an operator is shown in bold. Optional arguments are shown in [ ]. If the optional direction argument is not specified, the configuration is applied to all relevant directions. We use a default grain size of 256 for parallelization.

Apply Scheduling Functions	Descriptions
`program->configApplyDirection(label, config);`	Config options: **SparsePush**, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
`program->configApplyParallelization(label, config, [grainSize], [direction]);`	Config options: **serial**, dynamic-vertex-parallel, static-vertex-parallel, edge-aware-dynamic-vertex-parallel, edge-parallel
`program->configApplyDenseVertexSet(label, config, [vertexset], [direction])`	Vertexset options: **both**, src-vertexset, dst-vertexset Config Options: **bool-array**, bitvector
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	Config options: **fixed-vertex-count** or edge-aware-vertex-count
`program->configApplyNUMA(label, config, [direction]);`	Config options: **serial**, static-parallel, dynamic-parallel
`program->fuseFields({vect1, vect2, ...})`	Fuses multiple arrays into a single array of structs.
`program->fuseForLoop(label1, la bel2, fused_label)`	Fuses together multiple loops.
`program->fuseApplyFunctions(label1, label2, fused_func)`	Fuses together two edgeset apply operators. The fused apply operator replaces the first operator.

4 SCHEDULING LANGUAGE

After specifying the algorithm using the language described in Section 3, programmers can explore different combinations of optimizations using GraphIt's scheduling language. In this section, we describe the design of the scheduling language functions and demonstrate how they work with PageRankDelta.

4.1 Scheduling Language

We use labels (**`#label#`**) in algorithm specifications to identify the statements on which optimizations apply. Programmers can assign a label on the left side of a statement and later reference it in the scheduling language. Fig. 6 shows a simple schedule for the PageRankDelta implementation in Fig. 4. The programmer adds label **s1** to the edgeset operation statement. After the `schedule` keyword, the programmer can make a series of calls to scheduling functions.

```
30     ...
31     for i in 1:maxIters
32       #s1# edges.from(frontier).apply(updateEdge);
         ...
38     end
   ...
41 schedule:
42 program->configApplyDirection("s1", "DensePull-SparsePush");
```

Fig. 6. Scheduling PageRankDelta.

We designed GraphIt's scheduling language functions (shown in Table 3) to allow programmers to compose together edge traversal direction, frontier data structure, parallelization, cache, NUMA, vertex data layout, and program structure optimizations discussed in Section 2. The `configApplyDirection` function allows programmers to configure directions used for traversal. The programmer can use the `configDenseVertexSet` function to switch between bitvector and boolean array for source and destination vertexsets. The `configApplyNumSSG` function configures the number of segmented subgraphs and how the subgraphs are partitioned (fixed-vertex-count and edge-aware-vertex-count). Setting the right number of segments and partitioning configuration allows random accesses to be restricted to a NUMA node or last level cache with balanced load as described in Section 2. `configApplyNUMA` configures the segmented subgraphs to be executed in parallel with static or dynamic NUMA node assignment (static-parallel and dynamic-parallel), ensuring the random memory accesses are restricted to the local NUMA node, while maintaining good parallel scalability. Finally, vertex data vectors can be fused together into an array of structs with `fuseFields`.

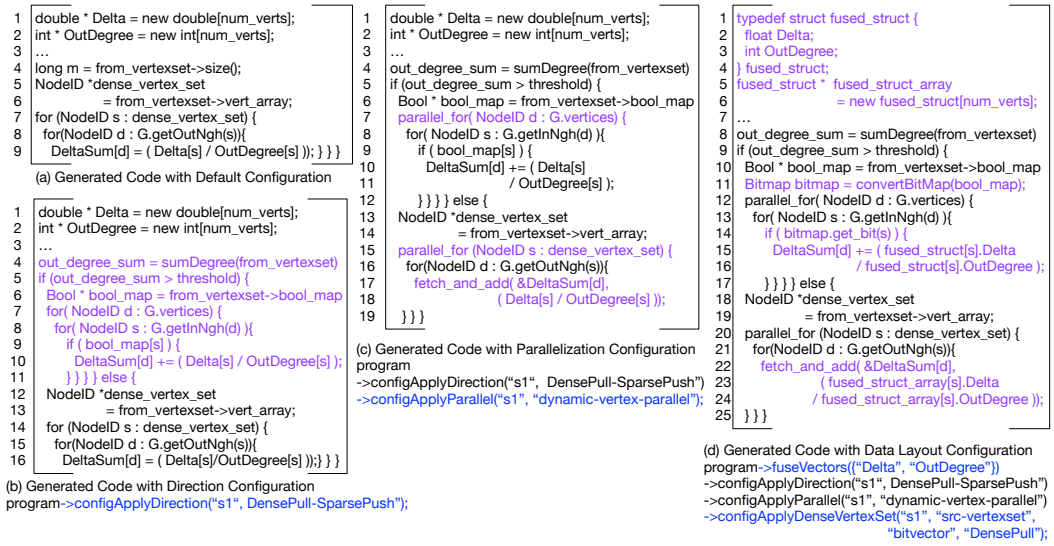

Fig. 7. Each subfigure shows pseudocode generated from applying the schedule in the caption to the GraphIt PageRankDelta code with labels from Fig. 4 and Fig. 6. The options in the caption highlighted in blue are newly added scheduling commands relative to the previous subfigure and the code highlighted in purple is pseudocode updated due to the new schedules.

To compose together different optimizations, the programmer first chooses a direction for traversal. Then the programmer can use the other scheduling functions to pick one option for the parallelization, graph partitioning, NUMA, and dense vertexset optimizations for the current direction. The programmer can configure each direction separately using the optional direction argument for hybrid directions (DensePush-SparsePush or DensePull-SparsePush). If no direction argument is specified, then the configuration applies to both directions.

4.2 Scheduling PageRankDelta

Fig. 7 shows different schedules for PageRankDelta. Fig. 7(a) starts with the pseudocode generated from the default schedule that performs a serial SparsePush traversal. Fig. 7(b) adds hybrid traversal code that first computes the sum of out-degrees and uses it to determine whether to do a DensePull or a SparsePush traversal. This allows the implementation to pick the traversal mode that minimizes the number of edges that need to be traversed, improving work-efficiency. Fig. 7(c) adds dynamic-vertex-parallelism to both directions in the generated code by parallelizing the loops and inserting synchronization code for SparsePush. Finally, Fig. 7(d) adds vertex data layout and bitvector optimizations. Fusing together the vectors Delta and OutDegree with the `fuseFields` function improves spatial locality of memory accesses since the two vectors are always accessed together. This optimization changes the declaration and access points for the arrays. Finally, for the DensePull direction, the source vertexset specified in `from` can be dynamically compressed into a bitvector to reduce the working set size, further improving spatial locality.

4.3 Scheduling Program Structure Optimizations

To support program structure optimizations, we introduce scoped labels, which allow labels to function even after complex program transformations, and scheduling functions for fusing together loops and edgeset `apply` operators. Fig. 8 shows two iterative edgeset `apply` operators (Lines 2 and 5) that can be fused together into a single iterative edgeset `apply` operator. GraphIt first performs loop fusion, creating a new loop (**l3**), and destroying the two old loops (**l1** and **l2**). Now, it would

be difficult if we wanted to schedule the first edgeset `apply` operator in the **l3** loop as the original loops **l1** and **l2** have been removed from the program. Since both edgeset `apply` operators have **s1** as their label, it is hard to identify them individually. To address this, we introduce scoping to labels. The two `apply` operators will obtain labels **l1:s1** and **l2:s1**, respectively.

We also need a name node, which enforces a named scope for the label. Loops **l1** and **l2** are replaced with name nodes with labels **l1** and **l2**, respectively. The resulting pseudocode is shown in Fig. 9. This enables the user to reference the first edgeset `apply` as **l3:l1:s1** and the second edgeset `apply` as **l3:l2:s1**. After the loops are fused together, we can use `fuseApplyFunctions` to create a new edgeset `apply` to replace the **l3:l1:s1** statement, which can be further configured (Fig. 10). The new edgeset `apply` function, `fusedFunc`, concatenates the statements in the original functions, `func1` and `func2`. In Section 7, we show that fusion of multiple iterative kernels with similar traversal patterns (Eigenvector Centrality and PageRank), and the vertex data vectors they access boosts the performance of the application by up to 60%.

```
#l1# for i in 1:10
    #s1# edges.apply(func1);
end
#l2# for i in 1:10
    #s1# edges.apply(func2);
end
schedule:
program->fuseForLoop("l1", "l2", "l3")
->fuseApplyFunctions("l3:l1:s1", "l3:l2:s1", "fusedFunc")
->configApplyDirection("l3:l1:s1", "DensePull");
```

Fig. 8. GraphIt loop and function fusion

```
#l3# for i in 1:10
    #l1# namenode
        #s1# edges.apply(func1);
    end
    #l2# namenode
        #s1# edges.apply(func2);
    end
end
```

Fig. 9. Pseudocode after loop fusion

```
#l3# for i in 1:10
    #l1# namenode
        #s1# edges.apply(fused_func);
    end
end
```

Fig. 10. Pseudocode after function fusion

5 SCHEDULING REPRESENTATION

The schedules for an optimized PageRankDelta implementation become even more complex than those shown in Fig. 7(d) as we further combine NUMA and cache optimizations. It is challenging to reason about the validity of, and to generate code for, combinations of optimizations. GraphIt relies on multiple scheduling representations, specifically the graph iteration space, the vertex data vector tags, and the scoped labels, to model combinations of edge traversal, vertex data layout, and program structure optimizations. Fig. 11 shows the full space of optimizations.

5.1 Graph Iteration Space

Motivation. The graph iteration space is an abstract model for edge traversals that represents the edge traversal optimizations specified by the scheduling commands in Table 3. The model simplifies the design of the compiler by representing different combinations of optimizations as multi-dimensional vectors. This representation enables the compiler to easily compose together different optimizations, reason about validity through dependence analysis, and generate nested loop traversal code. The graph iteration space also defines the space of edge traversal optimizations supported by GraphIt, revealing new combinations of optimizations not explored by prior work.
Definition. Let us assume that we have an operation that traverses edges and applies a user-defined function **F** on an edgeset Edges as shown in Fig. 12(a). A graph iteration space defines the set of directed edges on which **F** is applied and the strategy of traversing the edges. The concept of graph iteration space is inspired by the traditional iteration spaces in dense nested loops [Padua and Wolfe 1986; Wolf and Lam 1991]. First, we represent the graph as an adjacency matrix, where a column in a row has a value of one if the column represents a neighbor of the current row (the top part of Fig. 12(b)). With this representation, we can traverse through all edges using dense

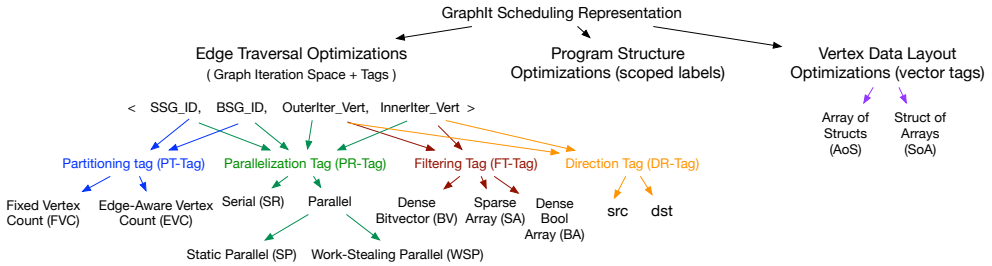

Fig. 11. GraphIt's scheduling representation for edge traversal, vertex data layout, and program structure optimizations. The tags of the graph iteration space represent the direction and performance optimization choices for each vertex data vector and each dimension of the graph iteration space.

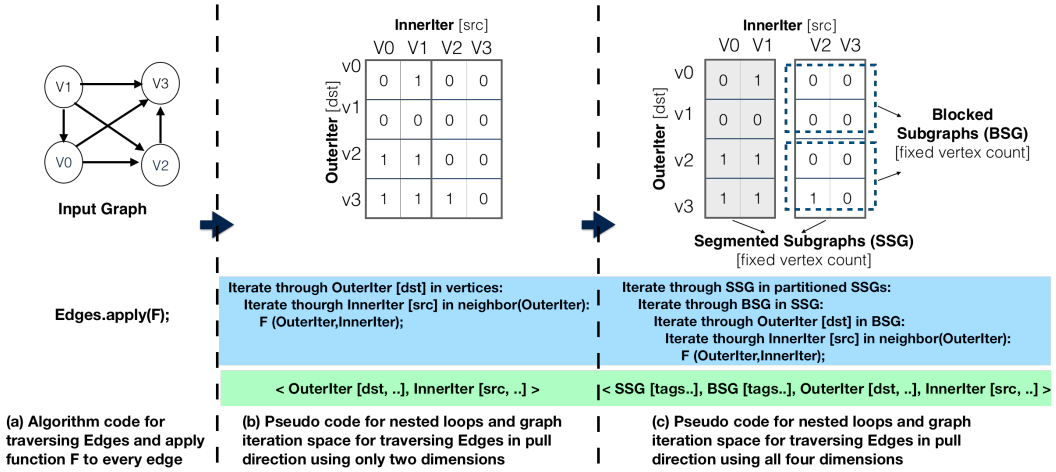

Fig. 12. Representing an edge traversal as nested loops and corresponding graph iteration spaces. Subfigure (c) shows the four dimensions for the graph iteration space, assuming using pull direction and the fixed vertex count partitioning strategy for both SSG and BSG dimensions.

$$\left\{ \langle\, S\ [\text{tags}],\ B\ [\text{tags}],\ O\ [\text{tags}],\ I\ [\text{tags}]\ \rangle \;\middle|\; \begin{array}{l} \text{src\_set = filtered } src \text{ vertexset of } \mathbf{F},\ \text{dst\_set = filtered } dst \text{ vertexset of } \mathbf{F} \\ O \in \text{src\_set} \wedge I \in \text{dst\_set or } O \in \text{dst\_set} \wedge I \in \text{src\_set} \\ \text{ssg\_set = subgraphs created by segmenting the graph based on InnerIter} \\ \text{bsg\_set = subgraphs created by blocking the graph based on OuterIter} \\ S \text{ (Segmented Subgraph ID)} \in \text{ssg\_set} \\ B \text{ (Blocked Subgraph ID)} \in \text{bsg\_set} \\ \langle\, O, I\, \rangle \in \text{edges within the subgraph } (B \text{ or } S) \end{array} \right\}$$

Fig. 13. Definition of the graph iteration space with four dimensions. S, B, O, and I are abbreviations for SSG_ID, BSG_ID, OuterIter, and InnerIter.

two-level nested for loops that iterate through every row and every column. The traversal can be viewed as a traditional 2-D iteration space. Unlike the dense iteration space, the edge traversal only happens when there is an edge from the source to the destination. Thus, we can eliminate unnecessary traversals and make the loops sparse by iterating only through columns with non-zero values in each row (the blue part in Fig. 12(b)). We define the row iterator variable as *OuterIter*, and the column iterator variable as *InnerIter*. The green part of Fig. 12(b) shows that a two dimensional graph iteration space vector is used to represent this two-level nested traversal. The two-level nested for loops can be further blocked and segmented into up to four dimensions as shown in Fig. 12(c). The dimensions of the graph iteration space encode the nesting level of the edge traversal, and the tags for each dimension specify the strategy used to iterate through that dimension. We provide more details of the graph iteration space below.

< S [PR-tag, (PT-Tag, count)], B [PR-tag, (PT-Tag, count)], O [DR-Tag, PR-Tag, FT-Tag], I [DR-Tag, PR-Tag, FT-Tag] >

Fig. 14. Graph Iteration Space Tags: Direction Tags (DR-Tag), Partitioning Tags (PT-Tag), Parallelization Tags (PR-Tag), and Filtering Tags (FT-Tag) (explained in Fig. 11) specify direction and optimization strategy for each dimension, and are shown in square brackets next to each dimension.

Table 4. Mapping between GraphIt's scheduling language functions to the relevant dimensions and tags (highlighted in bold) of the graph iteration space.

Apply Scheduling Functions	Graph Iteration Space Dimensions and Tags Configured
`program->configApplyDirection(label, config);`	⟨ *S* [tags], *B* [tags], ***O*** [**direction tag, filtering tag**], ***I*** [**direction tag, filtering tag**] ⟩. Note, for hybrid directions (e.g. DensePull-SparsePush), two graph iteration space vectors are created, one for each direction.
`program->configApplyParallelization(label, config, [grainSize], [direction]);`	⟨ *S* [tags], ***B*** [**partitioning tag, parallelization tag**], *O* [tags], *I* [tags] ⟩
`program->configApplyDenseVertexSet(label, config, [vertexset], [direction])`	⟨ *S* [tags], *B* [tags], ***O*** [**filtering tag**], ***I*** [**filtering tag**] ⟩
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	⟨ ***S*** [**partitioning tag**], *B* [tags], *O* [tags], *I* [tags] ⟩
`program->configApplyNUMA(label, config, [direction]);`	⟨ ***S*** [**parallelization tag**], *B* [tags], *O* [tags], *I* [tags] ⟩

Graph Iteration Space Dimensions. The graph iteration space in GraphIt uses four dimensions, defined in Fig. 13 and illustrated in Fig. 12. The dimensions are ⟨ SSG_ID, BSG_ID, OuterIter, InnerIter ⟩ and are abbreviated as ⟨ *S*, *B*, *O*, *I* ⟩. Unused dimensions are marked with ⊥.

OuterIter (*O*) and InnerIter (*I*) in Fig. 13 are the vertex IDs of an edge (Fig. 12(b)). The ranges of *O* and *I* dimensions depend on the direction. For the push direction, *O* is in the filtered source vertexset (src_set) and *I* is in the filtered destination vertexset (dst_set). For the pull direction, *O* is in the dst_set and *I* is in the src_set. The OuterIter dimension sequentially accesses vertices, while the InnerIter dimension has a random access pattern due to neighbor vertex IDs not being sequential. The edge (O, I) is in the edgeset of the subgraph identified by BSG_ID and SSG_ID.

The BSG_ID (Blocked Subgraph ID) dimension identifies a Blocked Subgraph (BSG) in the Blocked Subgraphs Set (bsg_set). The bsg_set is created by partitioning the graph by the OuterIter dimension as illustrated in the top part of Fig. 12(c). This partitioning transforms the loops that traverse the edges without changing the graph data structure. The graph can be partitioned with a grain size on the number of OuterIter vertices or on the total number of edges per BSG, depending on the schedule. This dimension controls the different strategies for parallelization optimizations.

The SSG_ID (Segmented Subgraph ID) identifies a Segmented Subgraph (SSG) in the Segmented Subgraphs Set (ssg_set). The ssg_set is created by partitioning the graph by the InnerIter dimension as demonstrated in the top part of Fig. 12(c). The partitioning transforms both the graph data structure and the loops that traverse the edges. Details of the partitioning scheme are described in prior work [Zhang et al. 2017]. This dimension controls the range of random accesses, enabling cache and NUMA optimizations. The ordering of the dimensions ensures that the graph is segmented into SSGs before each SSG is blocked into BSGs.

Graph Iteration Space Tags. Each dimension is annotated with tags to specify the direction and optimization strategies (Fig. 11 illustrates the tags in GraphIt). There are four types of tags: Direction Tags (DR-Tag), Partitioning Tags (PT-Tag), Parallelization Tags (PR-Tag), and Filtering Tags (FT-Tag). We show tags for each dimension within square brackets in Fig. 14. Table 4 shows the mapping between scheduling language commands from Section 4 and the corresponding graph iteration space vector and tags.

Direction Tags specify whether the traversal is in push or pull direction. In the push direction, the OuterIter is tagged as *src* and InnerIter tagged as *dst*; in the pull direction, the tags are reversed.

Table 5. The schedules applied for PageRankDelta and the generated graph iteration space vectors with tags, following the examples in Fig. 7. Newly added scheduling commands and the affected dimensions and tags in the graph iteration space are highlighted in bold. ⊥ is an unused dimension. The abbreviated dimensions and tags are defined in Fig. 13 and Fig. 11. The keyword 'Program' and the continuation symbol '->' are omitted. 'ca' is an abbreviation for 'configApply'. "caParallel" is short for configApplyParallelization. Note that configApplyNumSSG uses an integer parameter (X) which is dependent on the data and hardware system.

PageRankDelta Schedules	Graph Iteration Space
`caDirection("s1", "SparsePush");`	⟨ ⊥, ⊥, O [src, SR, SA], I [dst, SR] ⟩
`capDirection("s1",`**`"DensePull-SparsePush"`**`);`	runtime decision between two graph iteration space vectors ⟨ **⊥**, **⊥**, ***O*** **[*dst*, SR]**, ***I*** **[*src*, SR, BA]** ⟩ and ⟨ ⊥, ⊥, O [src, SR, SA], I [dst, SR] ⟩
`caDirection("s1","DensePull-SparsePush");` **`caParallel("s1","dynamic-vertex-parallel");`**	runtime decision between two graph iteration space vectors ⟨ ⊥, ***B*** **[WSP, (FVC, 1024)]**, O [dst, SR], I [src, SR, BA] ⟩ and ⟨ ⊥, ***B*** **[WSP, (FVC, 1024)]**, O [src, SR, SA], I [dst, SR] ⟩

`caDirection("s1","DensePull-SparsePush");`
`caParallel("s1","dynamic-vertex-parallel");`
`caDenseVertexSet("s1","src-vertexset",`
`"bitvector","DensePull");`
	runtime decision between two graph iteration space vectors ⟨ ⊥, B [WSP, (FVC, 1024)], O [dst, SR], I [src, SR, **BV**] ⟩ and ⟨ ⊥, B [WSP, (FVC, 1024)], O [src, SR, SA], I [dst, SR] ⟩

`caDirection("s1","DensePull-SparsePush");`
`caParallel("s1","dynamic-vertex-parallel");`
`caDenseVertexSet("s1","src-vertexset",`
`"bitvector","DensePull");`
`caNumSSG("s1","fixed-vertex-count",X,`
`"DensePull");`
	runtime decision between two graph iteration space vectors ⟨ S [**SR**, (**FVC, num_vert / X**)], B [WSP, (FVC, 1024)], O [dst, SR], I [src, SR, BV] ⟩ and ⟨ ⊥, B [WSP, (FVC, 1024)], O [src, SR, SA], I [dst, SR] ⟩

Partitioning Tags specify the strategy used for partitioning the SSG_ID or BSG_ID dimensions. For example, the default fixed vertex count (FVC) partitioning strategy will partition the graph based on a fixed number of InnerIter or OuterIter vertices as shown in Fig. 12(c). Depending on the input, this scheme may lead to an unbalanced number of edges in each SSG or BSG. Alternatively, the edge-aware vertex count (EVC) scheme partitions each subgraph with a different number of InnerIter or OuterIter vertices to ensure each subgraph have similar number of edges. The EVC tag is used when the users specify the edge-aware-dynamic-vertex-parallel option with `configApplyParalllelization` or the edge-aware-vertex-count option with `configApplyNumSSG`.

Parallelization Tags control whether to iterate through the dimension using serial (SR), static-partitioned parallel (SP), or dynamic work-stealing parallel (WSP) execution strategies. The PR-Tag for the BSG_ID dimension controls the parallelization strategy across different Blocked Subgraphs within a Segmented Subgraph. Tagging the SSG_ID dimension to be parallel enables NUMA optimizations by executing multiple SSGs in different sockets in parallel. If work-stealing is enabled, threads on one socket can steal unprocessed SSGs from another socket to improve load balance.

Filtering Tags on the OuterIter and InnerIter dimensions control the underlying data structure. Filtering is implemented with sparse arrays (SA), dense boolean arrays (BA), or dense bitvectors (BV). The sparse arrays contain all of the vertices that pass the filtering, while the dense boolean arrays or the bitvectors set the value to true or the bit to one for each vertex that passes the filtering.

Graph Iteration Spaces for PageRankDelta. Table 5 continues to use PageRankDelta as an example to illustrate how scheduling language commands generate graph iteration space vectors and tags. The first row shows that the SparsePush schedule maps to a graph iteration space vector with only two dimensions used (⊥ means the dimension is unused). The direction tags for the OuterIter and InnerIter dimensions, *src* and *dst*, indicate that this graph iteration space is for the push direction. Going from SparsePush to DensePull-SparsePush creates a new graph iteration space vector for the pull direction. A runtime threshold on the size of the source vertexset is used to decide which vector gets executed. The `configApplyParallelization` function configures the

```
# generate edgeset traversal code based on
# apply_expr: algorithmic operators. such as apply, from, to, src_filter, and dst_filter
# gis_vec_list: graph iteration space vectors generated with the scheduling language
gen-edgeset-apply (List<GraphIterSpaceVector> gis_vec_list, EdgesetApplyExpr apply_expr){
    # two graph iteration space vectors might be supplied
    # one of the two graph iteration space vectors will be selected at runtime
    if (gis_vec_list contains two graph iteration space vectors) {
        # generate a condition to select one of the two graph iteration space vectors
        print "if"; emit-gis-vector-select-condition (gis_vec_list, apply_expr); print "{";
        gen-SSG (gis_vec_list[0], apply_expr);
        print " } else { ";
        gen-SSG (gis_vec_list[1], apply_expr);
        print "}";
    } else { gen-SSG (gis_vec_list[0], apply_expr); }
}
# generate traversal code for the SSG dimension
gen-SSG (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    emit-SSG-traversal-loop (gis_vec, apply_expr); # see subsection SSG Code Generation
    gen-BSG (gis_vec, apply_expr);
    emit-SSG-post-traversal-code (gis_vec, apply_expr);
}
# generate traversal code for the BSG dimension
gen-BSG (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    emit-BSG-traversal-loop (gis_vec, apply_expr); # see subsection BSG Code Generation
    gen-OuterIter-InnterIter (gis_vec, apply_expr);
    emit-BSG-post-traversal-code (gis_vec, apply_expr);
}
# generate traversal code for the outerIter and innerIter
gen-OuterIter-InnterIter (GraphIterSpaceVector gis_vec, EdgesetApplyExpr apply_expr) {
    # see subsection OuterIter and InnerIter Code Generation
    emit-OuterIter-InnerIter-Nested-loops (gis_vec, apply_expr);
}
```

Fig. 15. Code generation algorithm for the graph iteration space.

BSG_ID dimension with work-stealing parallelism (WSP) and uses the default 1024 grainsize. The fourth row demonstrates that `configDenseVertexSet` sets the filtering tag for the innerIter dimension to bitvector (BV) in the graph iteration space vector for the pull direction. Finally, `configNumSSG` sets up the SSG_ID dimension to partition the graph for cache locality. In the fixed-vertex-count configuration (FVC), the InnerIter range for each SSG is computed by dividing the total number of vertices by the number of SSGs specified with X.

Generalizing Graph Iteration Spaces. The graph iteration space concept can be generalized to expand the space of supported optimizations. In GraphIt, we restrict the graph iteration space to four dimensions with fixed partitioning schemes. Adding more dimensions and/or removing constraints on how the dimensions are partitioned can potentially represent additional optimizations.

5.2 Vertex Data Layout and Program Structure Optimizations Representation

Since the vertex data are stored as abstract vectors, they can be implemented as an array of structs or struct of arrays. We use vector tags to tag each vertex data vector as Array of Structs (AoS) or a separate array in the implicit global struct (SoA). These tags can be configured with the `fuseFields` scheduling function. Program structure optimizations update the structure of the loops and edgeset `apply` operators. We use the scoped labels (described in Section 4.3), which are specified in the scheduling language with `fuseForLoop` and `fuseApplyFunctions`, to represent the optimizations.

6 COMPILER IMPLEMENTATION

This section describes the GraphIt compiler, which generates optimized C++ code from an algorithm and a schedule. We also built an autotuner on top of the compiler to automatically find high-performance schedules.

```
for (int i = 0; i < Frontier.size(); i++){
  NodeID src = Frontier.vert_array[i];
  for (NodeID dst : G.getOutNghs(src)){
    if (dstFunc(dst)){
      applyFunc(src, dst); }}}
```

Fig. 17. Generated SparsePush code.

```
for (NodeID dst = 0; dst < num_verts; dst++){
  if (dstFunc){
    for (NodeID src : G.getInNghs(dst)){
      if (Frontier.bool_map(src)){
        applyFunc(src, dst); }}}}
```

Fig. 18. Generated DensePull code.

6.1 Code Generation for Graph Iteration Space

We first show the high-level code generation algorithm for the graph iteration space in Fig. 15. To deal with hybrid traversal modes that have two graph iteration space vectors, such as DensePull-SparsePush, `gen-edgeset-apply` generates two implementations of edge traversal logic with additional logic to choose an implementation based on the sum of the out-degrees of active vertices (the "if", "else", and `emit-gis-vector-select-condition` shown in Fig. 15) as described in Section 2. The functions `gen-SSG`, `gen-BSG`, and `gen-OuterIter-InnerIter` generate nested traversal loops for the different graph iteration space dimensions. Below, we provide more details on the code generation functions and the mechanisms to ensure the validity of the optimizations.

OuterIter and InnerIter Code Generation. We demonstrate how to generate traversal code for the OuterIter and InnerIter dimensions using a simple example with the SparsePush configuration shown in Fig. 16 (graph iteration space and tags: $\langle \perp, \perp, O\ [src, \text{SR}, \text{SA}], I\ [dst, \text{SR}, \text{BA}] \rangle$; abbreviations and sets are defined in Fig. 13 and Fig. 11).

```
#s1# edges.from(Frontier).dstFilter(dstFunc).apply(applyFunc)
schedule:
program->configApplyDirection("s1","SparsePush");
```

Fig. 16. SparsePush configuration.

For the push direction, OuterIter is *src* and InnerIter is *dst*. Since the source (OuterIter) filtering is tagged as Sparse Array (SA), the outer loop iterates over the source vertexset (Frontier). The `dst` (InnerIter) filtering uses the user-defined boolean function `dstFunc`. The generated code is shown in Fig. 17.

We show code generated for a DensePull traversal mode ($\langle \perp, \perp, O\ [dst, \text{SR}, \text{BA}], I\ [\text{src}, \text{SR}, \text{BA}] \rangle$) in Fig. 18. The OuterIter is now *dst* and the InnerIter is *src*. The user-defined function `applyFunc` is applied to every edge as before. The vertexsets are automatically converted from the sparse array of vertices (`vert_array` shown in the SparsePush example above) to a boolean map (`bool_map` in the DensePull example). Filtering on destination vertices (`dstFilter`) is attached as an `if` statement next to the *dst* iterator (OuterIter).

Blocked Subgraph (BSG) Code Generation. The BSG dimension in the graph iteration space is created by partitioning the OuterIter dimension. GraphIt uses the partitioning tag for this dimension to control the granularity and blocking strategy for load balancing, and the parallelization tag to control the mode of parallelization. Fig. 19 shows an example of the generated code, assuming OuterIter represents *src*.

```
parallel_for (int BSG_ID = 0; BSG_ID < g.num_chunks; BSG_ID++){
  for (NodeID src = g.chunk_start[BSG_ID]; src < g.chunk_end[BSG_ID]; src++)
    for (NodeID dst : G.getOutNghs(src))
      applyFunc(src,dst);}
```

Fig. 19. Generated BSG code.

If the edge-aware vertex count (EVC) partitioning tag is used, the compiler generates chunks with approximately the number of edges specified by the schedule. For the fixed vertex count (FVC) partitioning tag, the compiler uses the built-in grain size in OpenMP. For parallelization tags static parallelism (SP) and dynamic work-stealing parallelism (WSP), we simply use the OpenMP pragmas to implement `parallel_for` (`pragma omp for parallel schedule (static)` and `schedule (dynamic)`).

Segmented Subgraph (SSG) Code Generation. Using the SSG dimension requires adding a loop outside of the existing traversals and changing the data layout of the graph. GraphIt generates code in the main function to create the SSGs by partitioning the graph by InnerIter. This partitioning

can use a fixed range of vertices (FVC) in the InnerIter or a flexible range of vertices that takes into account the number of edges in each SSG (EVC) with an edge grain size. The random memory access range in each SSG is restricted to improve locality. Fig. 20 shows edge traversal code that uses both SSG and BSG dimensions (⟨ S [SR], B [SR], O [dst, SR], I [src, SR] ⟩). The segmented subgraphs are stored in `g.SSG_list`.

```
1 for (int SSG_ID = 0; SSG_ID < num_SSG; SSG_ID++){
2   sg = g.SSG_list[SSG_ID];
3   for (int BSG_ID = 0; BSG_ID < sg.num_chunks; BSG_ID++){
4     for (NodeID dst = sg.chunk_start[BSG_ID]; dst < sg.chunk_end[BSG_ID]; dst++)
5       for (NodeID src : G.getInNghs(dst))
6         applyFunc(src,dst);}}
```

Fig. 20. Generated SSG and BSG code.

The cache optimization processes one SSG at a time (SR), but processes the BSGs within the SSG in parallel. The programmer can enable NUMA optimizations by specifying the parallelization tag for SSG as static parallel (SP); the compiler then assigns different SSGs to be executed on different sockets. GraphIt implements this assignment using `numa_alloc` in the main function to first allocate SSGs on different sockets, and then uses the `proc_bind` API in OpenMP to assign threads to process each socket-local subgraph. If work-stealing parallelism (WSP) is enabled for SSGs, then a socket can steal an SSG allocated on another socket if no work remains on the current socket.

In some cases, we need to use NUMA-local buffers to store intermediate results from each SSG. The compiler generates code for allocating NUMA-local buffers and changes the data references from updating global vertex data vectors to the NUMA-local buffers. The compiler also generates code for a merge phase that merges NUMA-local buffers to update the global data vectors.

Validity of Optimizations. GraphIt ensures the validity of single edge traversal optimizations by imposing a set of restrictions on the GraphIt language and using dependence analysis to insert appropriate atomic synchronization instructions.

We enforce some restrictions on read-write accesses and reduction operators for vertex data vectors across user-defined functions used in `srcFilter`, `dstFilter`, and edgeset `apply` functions for a given edgeset traversal operation. Each vertex data vector must have only one of the following properties: read-only, write-only, or reduction. Additionally, reductions are commutative and associative. With these two restrictions, transformations do not need to preserve read-after-write dependences, and transformations remain valid independent of edge traversal order. Therefore, a transformed program is valid as long as each filtered edge is processed exactly once. GraphIt does provide `asyncMax` and `asyncMin` reduction operators for cases where there can be some read and write dependences. The programmer is responsible for ensuring that reordering traversals will not affect the final outcome of the program when using `asyncMax` and `asyncMin`. These operators are useful for applications such as connected components and single-source shortest paths as the vertex data values will eventually converge.

To ensure that each filtered edge is processed exactly once, we insert synchronization code to vertex data vector updates by leveraging dependence analysis theory from dense loop iteration spaces [Li et al. 1989; Maydan et al. 1991]. Dependence analysis is well-suited for GraphIt because the language prevents aliasing and each vertex data vector represents a separate data structure. Additionally, the goal of the analysis is not to automatically parallelize the loop with a correctness guarantee, but the much easier task of determining whether synchronization code is necessary for a given parallelization scheme. Accomplishing this task does not require a precise distance vector.

Below we show a code snippet of PageRankDelta with the SparsePush configuration (⟨ ⊥, ⊥, O [src, SR, SA], I [dst, SR] ⟩), the distance vector, and read-write properties of the vectors.

```
1 for (int i = 0; i < Frontier.size(); i++){
2   NodeID src = Frontier.vert_array[i];
3   for (NodeID dst : G.getOutNghs(src)){
4       DeltaSum[dst] += Delta[src]/OutDegree[src];
5       }}}
```

Vector Dependence	Distance Vector	Read-Write
DeltaSum	$\langle *, 0 \rangle$	reduction
Delta	$\langle 0, 0 \rangle$	read-only
OutDegree	$\langle 0, 0 \rangle$	read-only

The compiler builds a dependence vector for the OuterIter and InnerIter dimensions based on their direction tags. We see that DeltaSum has a dependence with the reduction operator (it is both read from and written to). Different source nodes can update the same *dst*, and so we assign $*$ to the first element of the distance vector to denote that there is a dependence on an unknown iteration of *src*, which maps to OuterIter based on the direction tags. Given a *src*, we know that the *dst*'s are all different, and thus, there is no data dependence on the second iterator and we assign the second value of the distance vector as 0. Since Delta and OutDegree are both read-only, they have the distance vector $\langle 0, 0 \rangle$ with no dependence across different iterations. Given that DeltaSum's distance vector's first element is $*$, the compiler knows that synchronization must be provided when parallelizing the OuterIter (outer loop). If only the InnerIter (inner loop) is parallelized, then no synchronization is needed.

A similar analysis works on a DensePull ($\langle \perp, \perp, O$ [*dst*, SR, BA], I [*src*, SR] $\rangle$) PageRankDelta. The code snippet and distance vectors are shown below. The first element in the distance vector for DeltaSum is 0 given that there is no dependence among different destination vertices and OuterIter represents *dst*. However, the value is $*$ on the second element because different sources will update the same destination. Parallelizing OuterIter in this case does not require any synchronization.

```
1 for (NodeID dst = 0; dst < num_verts; dst++) {
2   for (NodeID src : G.getInNghs(dst)){
3       if (Frontier.bool_map(src))
4           DeltaSum[dst] += Delta[src]/OutDegree[src];
5           }}}
```

Vector Dependence	Distance Vector	Read Write
DeltaSum	$\langle 0, * \rangle$	reduction
Delta	$\langle 0, 0 \rangle$	read-only
OutDegree	$\langle 0, 0 \rangle$	read-only

Since BSG is partitioned by OuterIter, parallelizing the BSG_ID dimension would have the same effect as parallelizing OuterIter. Similarly, parallelizing SSG_ID has the same effect as parallelizing InnerIter given SSG is partitioned by InnerIter.

When applying NUMA optimizations to the second code snippet with the DensePull direction (parallelizing both the SSG and BSG dimensions), we have a dependence vector of $\langle *, * \rangle$ for DeltaSum. In this case, GraphIt writes the updates to DeltaSum[*dst*] to a socket-local buffer first and later merges buffers from all sockets to provide synchronization.

For the hybrid traversal configurations, GraphIt generates two versions of the user-defined `apply` function since the synchronization requirements for the push and pull directions are different. Each version will be used in the corresponding traversal mode.

6.2 Code Generation for Vertex Data Layout Optimizations

To generate code with different physical data layouts for the vertex data (array of structs or struct of arrays), GraphIt generates declaration and initialization code in the main function and updates references to vertex data in the other functions. The compiler first transforms assignments on vertex data vectors into vertexset `apply` operations that set the values of the data vectors. If the programmer specifies the `fuseField` command, GraphIt generates a new struct type, an array of structs declaration, and changes the references to the vertex data vectors in the functions to access fields of the struct instead of separate arrays. For example, the assignment statement for the parent vector `parent : vector {Vertex}(int) = -1;` is implemented by first declaring an `apply` function `vertexset_apply_f` and another `vertices.apply` statement in the main

```
1 func vertexset_apply_f(v:Vertex)
2   parent[v] = -1;
3 end
4 func main()
5   vertices.apply(vertexset_apply_f);
6 end
```

Fig. 21. Vertexset `apply` code.

function that uses `vertexset_apply_f` as shown in Fig. 21. The vector access expression in the `apply` function will then be lowered from `parent[v]` to `fused_struct[v].parent` (Line 2 of Fig. 21). The correctness of the program is not impacted by the vertex data layout optimization as it does not affect the execution ordering of the program.

6.3 Code Generation for Program Structure Optimizations

Traditional compilers with a fixed number and order of optimization passes are ill-suited for program structure optimizations, such as kernel fusion. The GraphIt compiler introduces a new schedule-driven optimization pass orchestration design that allows users to add more optimization passes and dictate the order of the added optimizations with the label-based scheduling language described in Section 4. Users can perform fine-grained loop fusion, loop splitting, and fusion of `apply` functions on loops and functions specified with statement labels and scheduling commands. These optimizations are implemented as customized optimization passes on the intermediate representation of the program. GraphIt implements these program structure transformation schedules by adding new optimization passes that transform the intermediate representation.

6.4 Autotuning GraphIt Schedules

Finding the right set of schedules can be challenging for non-experts. GraphIt can have up to 10^5 valid schedules with each run taking more than 30 seconds for our set of applications and input graphs. Exhaustive searches would require weeks of time. As a result, we use OpenTuner [Ansel et al. 2014] to build an autotuner on top of GraphIt that leverages stochastic search techniques (e.g., AUC bandit, greedy mutation, differential mutation, and hill climbing) to find high-performance schedules within a reasonable amount of time.

Search space. We limit the tuning to a single edgeset `apply` operation identified by the user. The autotuner will try different configurations for the direction of the traversal (`configApplyDirection`), the parallelization scheme (`configApplyParallelization`), the data layout for the dense vertexset (`configApplyDenseVertexSet`), the partitioning strategy of the graph (`configApplyNumSSG`), and the NUMA execution policy (`configApplyNUMA`).

Not all generated schedules are valid because schedules have dependencies among them. For example, `configApplyNumSSG`, which takes a direction parameter, is only valid if the direction specified is also set by `configApplyDirection`. Instead of reporting an invalid schedule as error, GraphIt's autotuner ignores invalid schedules to smooth the search space. For example, the `configApplyNumSSG` configuration is ignored if the direction specified is not valid.

7 EVALUATION

In this section, we compare GraphIt's performance to state-of-the-art frameworks and DSLs on graphs of various sizes and structures. We also analyze performance tradeoffs among different GraphIt schedules. We use a dual socket system with Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads. The system has 128GB of DDR3-1600 memory and 30 MB last level cache on each socket, and runs with Transparent Huge Pages (THP) enabled.

Data Sets. Table 6 lists our input datasets and their sizes. LiveJournal, Twitter, and Friendster are three social network graphs. Friendster is special because its number of edges does not fit into a 32-bit signed integer. We use WebGraph from the 2012 common crawl. USAroad is a mesh network with small and undeviating degrees. The Netflix rating dataset and its synthesized expansion (Netflix2x) are used to evaluate Collaborative Filtering.

Algorithms. We try to use the same algorithms across different frameworks to study the impact of performance optimizations. Our evaluation is done on seven algorithms: PageRank (PR), Breadth-First Search (BFS), Connected Components (CC) with synchronous label propagation, Single Source Shortest Paths (SSSP) with frontier based Bellman-Ford algorithm, Collaborative Filtering (CF), Betweenness Centrality (BC), and PageRankDelta (PRDelta). For Galois, we used the asynchronous algorithm for BFS, and the Ligra algorithm for SSSP.

Existing Frameworks. We compare GraphIt's performance to six state-of-the-art in-memory graph processing systems: Ligra, GraphMat, Green-Marl, Galois, Gemini, and Grazelle. Ligra has fast implementations of BFS and SSSP [Shun and Blelloch 2013]. Among prior work, GraphMat has the fastest shared-memory implementation of CF [Sundaram et al. 2015]. Green-Marl is one of the fastest DSLs for the algorithms we evaluate [Hong et al. 2012]. Galois (v2.2.1) has an efficient asynchronous engine that works well on road graphs [Nguyen et al. 2013]. Gemini is a distributed graph processing system with notable shared-machine performance [Zhu et al. 2016]. Compared to existing frameworks, Grazelle has the fastest PR and CC using edge list vectorization, inner loop parallelism, and NUMA optimizations [Grossman et al. 2018].

Table 6. Graphs used for experiments. LiveJournal and Friendster are undirected, and the number of edges counts each edge in both directions. All other graphs are directed.

Dataset	Num. Vertices	Num. Edges
LiveJournal (LJ) [Davis and Hu 2011]	5 M	69 M
Twitter (TW) [Kwak et al. 2010]	41 M	1469 M
WebGraph (WB) [Meusel et al. [n. d.]]	101 M	2043 M
USAroad (RD) [Demetrescu et al. [n. d.]]	24 M	58 M
Friendster (FT) [Leskovec and Krevl 2014]	65.6 M	3.6 B
Netflix (NX) [Bennett et al. 2007]	0.5 M	198 M
Netflix2x (NX2) [Li et al. 2013]	1 M	792 M

7.1 Comparisons with State-of-the-Art Frameworks

Table 7 shows the execution time of GraphIt and other systems. The best performing schedules for GraphIt are shown in Table 8. Table 9 shows the line counts of four graph algorithms for each framework. GraphIt often uses significantly fewer lines of code compared to the other frameworks. Unlike GraphIt, other frameworks with direction optimizations require programmers to provide many low-level implementation details as discussed in Section 3. GraphIt outperforms the next fastest of the six state-of-the-art shared-memory frameworks on 24 out of 32 experiments by up to 4.8×, and is never more than 43% slower than the fastest framework on the other experiments.

PR. GraphIt has the fastest PR on 4 out of the 5 graphs and is up to 54% faster than the next fastest framework because it enables both cache and NUMA optimizations when necessary. Table 10 shows that on the Twitter graph, GraphIt has the lowest LLC misses, QPI traffic, and cycles stalled compared to Gemini and Grazelle, which are the second and third fastest. GraphIt also reduces the line count by up to an order of magnitude compared to Grazelle and Gemini as shown in Table 9. Grazelle uses the Vector-Sparse edge list to improve vectorization, which works well on graphs with low-degree vertices [Grossman et al. 2018], outperforming GraphIt by 23% on USAroad. GraphIt does not yet have this optimization, but we plan to include it in the future. Frameworks other than Gemini and Grazelle do not optimize for cache or NUMA, resulting in much worse running times.

BFS. GraphIt has the fastest BFS on 4 out of the 5 graphs (up to 28% faster than the next fastest) because of its ability to generate code with different direction and bitvector optimizations. On LiveJournal, Twitter, WebGraph, and Friendster, GraphIt adopts Ligra's direction optimization. On USAroad, GraphIt always uses SparsePush and omits the check for when to switch traversal direction, reducing runtime overhead. In the pull direction traversals, GraphIt uses bitvectors to represent the frontiers when boolean array representations do not fit in the last level cache, whereas Ligra always uses boolean arrays and Grazelle always uses bitvectors. GraphIt outperforms Galois' BFS, even though Galois is highly-optimized for road graphs. GraphMat and Green-Marl do not

Table 7. Running time (seconds) of GraphIt and state-of-the-art frameworks. The fastest results are bolded. The missing numbers correspond to a framework not supporting an algorithm and/or not successfully running on an input graph. Galois' Betweenness Centrality (BC) uses an asynchronous algorithm, while other frameworks use a synchronous one. We ran PageRank (PR) for 20 iterations, PageRankDelta (PRDelta) for 10 iterations, and Collaborative Filtering (CF) for 10 iterations. Breadth-First Search (BFS), Single Source Shortest Paths (SSSP), and Betweenness Centrality (BC) times are averaged over 10 starting points.

Algorithm	PR					BFS					CC					CF	
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	NX	NX2
GraphIt	**0.342**	**8.707**	**16.393**	0.909	**32.571**	0.035	**0.298**	**0.645**	**0.216**	**0.490**	0.068	**0.890**	**1.960**	17.100	**2.630**	**1.286**	**4.588**
Ligra	1.190	49.000	68.100	1.990	201.000	**0.027**	0.336	0.915	1.041	0.677	**0.061**	2.780	5.810	25.900	13.000	5.350	25.500
GraphMat	0.560	20.400	35.000	1.190		0.100	2.800	4.800	1.960		0.365	9.8	17.9	84.5		5.010	21.600
Green-Marl	0.516	21.039	42.482	0.931		0.049	1.798	1.830	0.529		0.187	5.142	11.676	107.933			
Galois	2.788	30.751	46.270	9.607	117.468	0.038	1.339	1.183	0.220	3.440	0.125	5.055	15.823	12.658	18.541		
Gemini	0.430	10.980	16.440	1.100	44.600	0.060	0.490	0.980	10.550	0.730	0.150	3.850	9.660	85.000	13.772		
Grazelle	0.368	15.700	20.650	**0.740**	54.360	0.052	0.348	0.828	1.788	0.512	0.084	1.730	3.208	**12.200**	5.880		

Algorithm	SSSP					PRDelta					BC				
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT
GraphIt	0.055	**1.349**	**1.680**	**0.285**	**4.302**	**0.183**	**4.720**	**7.143**	**0.494**	**12.576**	0.102	1.550	2.500	**0.650**	**3.750**
Ligra	**0.051**	1.554	1.895	1.301	11.933	0.239	9.190	19.300	0.691	40.800	0.087	1.931	3.619	2.530	6.160
GraphMat	0.095	2.200	5.000	43.000											
Green-Marl	0.093	1.922	4.265	93.495							**0.082**	3.600	6.400	29.050	
Galois	0.091	1.941	2.290	0.926	4.643						0.237	3.398	4.289	0.806	9.897
Gemini	0.080	1.360	2.800	7.420	6.147						0.149	**1.358**	**2.299**	31.055	3.849

Table 8. Schedules that GraphIt uses for all applications on different graphs. The schedules assume that the edgeset apply operator is labeled with s1. The keyword 'Program' and the continuation symbol '->' are omitted. 'ca' is the abbreviation for 'configApply'. Note that configApplyNumSSG uses an integer parameter (X) which is dependent on the graph size and the cache size of a system. BC has two edgeset apply operators, denoted with s1 and s2.

Apps	USAroad	LiveJournal	Twitter, WebGraph, Friendster
PR	caDirection("s1", "DensePull") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePull") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePull") caParallelization("s1", "edge-aware-dynamic-vertex-parallel") caNumSSG("s1", "fixed-vertex-count", X) caNUMA("s1", "dynamic-static-parallel")
BFS	caDirection("s1", "SparsePush") caParallelization("s1", "static-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caDenseVertexSet("s1", "src-vertexset", "bitvector", "DensePull") caParallelization("s1", "dynamic-vertex-parallel")
CC	caDirection("s1", "DensePush-SparsePush") caParallelization("s1", "static-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caDenseVertexSet("s1", "src-vertexset", "bitvector", "DensePull") caParallelization("s1", "dynamic-vertex-parallel") caNumSSG("s1", "fixed-vertex-count", X, "DensePull")
SSSP	caDirection("s1", "SparsePush") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePush-SparsePush") caParallelization("s1", "dynamic-vertex-parallel")	caDirection("s1", "DensePush-SparsePush") caParallelization("s1", "dynamic-vertex-parallel")
PRDelta	caDirection("s1", "SparsePush") caParallelization("s1", "dynamic-vertex-parallel") fuseFields("OutDegree", "Delta")	caDirection("s1", "DensePull-SparsePush") caParallelization("s1", "dynamic-vertex-parallel") fuseFields("OutDegree", "Delta")	caDirection("s1", "DensePull-SparsePush") caDenseVertexSet("s1", "src-vertexset", "bitvector", "DensePull") caParallelization("s1", "dynamic-vertex-parallel") caNumSSG("s1", "fixed-vertex-count", X, "DensePull") caNUMA("s1", "static-parallel", "DensePull") fuseFields("OutDegree", "Delta")
BC	caDirection("s1", "SparsePush") caParallelization("s1", "static-vertex-parallel") caDirection("s2", "SparsePush") caParallelization("s2", "static-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caParallelization("s1", "dynamic-vertex-parallel") caDirection("s2", "DensePull-SparsePush") caParallelization("s2", "dynamic-vertex-parallel")	caDirection("s1", "DensePull-SparsePush") caDenseVertexSet("s1", "src-vertexset", "bitvector", "DensePull") caParallelization("s1", "dynamic-vertex-parallel") caDirection("s2", "DensePull-SparsePush") caDenseVertexSet("s2", "src-vertexset", "bitvector", "DensePull") caParallelization("s2", "dynamic-vertex-parallel")
CF	For Netflix and Netflix2x graphs caDirection("s1", "DensePull") caParallelization("s1", "edge-aware-dynamic-vertex-parallel") caNumSSG("s1", fixed-vertex-count, X)		

Table 9. Line counts of PR, BFS, CC, and SSSP for GraphIt, Ligra, GraphMat, Green-Marl, Galois, Gemini, and Grazelle. Only Green-Marl has fewer lines of code than GraphIt. GraphIt has an order of magnitude fewer lines of code than Grazelle (the second fastest framework on the majority of the algorithms we measured). For Galois, we only included the code for the specific algorithm that we used. Green-Marl has a built-in BFS.

	GraphIt	Ligra	GraphMat	Green-Marl	Galois	Gemini	Grazelle
PR	34	74	140	20	114	127	388
BFS	22	30	137	1	58	110	471
CC	22	44	90	25	94	109	659
SSSP	25	60	124	30	88	104	

have the direction optimization so it is much slower. Ligra is slightly faster than GraphIt on the smaller LiveJournal graph due to better memory utilization, but is slower on larger graphs.

Table 10. LLC miss rate, QPI traffic, cycles with pending memory loads and cache misses, and parallel running time (seconds) of PR, CC, and PRDelta running on Twitter, and CF running on Netflix.

Algorithm	PR				CC				PRDelta		CF	
Metrics	GraphIt	Ligra	Gemini	Grazelle	GraphIt	Ligra	Gemini	Grazelle	GraphIt	Ligra	GraphIt	Ligra
LLC miss rate (%)	**24.59**	60.97	45.09	56.68	**10.27**	48.92	43.46	56.24	**32.96**	71.16	**2.82**	37.86
QPI traffic (GB/s)	**7.26**	34.83	8.00	20.50	19.81	27.63	**6.20**	18.96	**8.50**	33.46	**5.68**	19.64
Cycle stalls (trillions)	**2.40**	17.00	3.50	4.70	**0.20**	0.96	1.20	0.30	**1.25**	5.00	**0.09**	0.22
Runtime (s)	**8.71**	49.00	10.98	15.70	**0.89**	2.78	3.85	1.73	**4.72**	9.19	**1.29**	5.35

CC. GraphIt has the fastest CC on Twitter, WebGraph, and Friendster because of the direction, bitvector, and cache optimizations. Table 10 shows GraphIt's reduced LLC miss rate and cycles stalled. Interestingly, Gemini has the lowest QPI traffic, but is much slower than GraphIt. With NUMA optimizations, vertices in one socket fail to see the newly propagated labels from vertices in another socket, resulting in slower convergence. Unlike other NUMA-aware graph processing frameworks, GraphIt can easily enable or disable NUMA optimizations depending on the algorithm. We choose the label propagation algorithm option on Galois and use the FRONTIERS_WITHOUT_ASYNC option on Grazelle in order to compare the same algorithm across frameworks. Galois' CC is 35% faster than GraphIt on USAroad because it uses a special asynchronous engine instead of a frontier-based model. We also ran Galois's union-find CC implementation but found it to be slower than GraphIt on all graphs except USAroad. Grazelle's CC using the Vector-Sparse format, implemented with hundreds of lines of assembly code as shown in Table 9, is 43% faster than GraphIt on USAroad. The best performing schedule that we found on USAroad without any asynchronous mechanism is DensePush-SparsePush.

CF. For CF, GraphIt is faster than Ligra and GraphMat (by 4–4.8×) because the edge-aware-dynamic-vertex-parallel schedule achieves good load balance on Netflix. Cache optimization further improves GraphIt's performance and is especially beneficial on Netflix2x.

SSSP. GraphIt has the fastest SSSP on 4 out of the 5 graphs because of its ability to enable or disable the direction optimization and the bitvector representation of the frontier. We run Galois with the Bellman-Ford algorithm so that the algorithms are the same across systems. We also tried Galois's asynchronous SSSP but found it to be faster than GraphIt only on WebGraph. Green-Marl's SSSP on USAroad is 328 times slower than GraphIt because it uses the DensePush configuration. On every round, it must iterate through all vertices to check if they are active. This is expensive on USAroad because for over 6000 rounds, the active vertices count is less than 0.4% of all the vertices.

PRDelta. GraphIt outperforms Ligra on all graphs by 2–4× due to better locality from using bitvectors as frontiers, fusing the Delta and OutDegree arrays as shown in Fig. 4, and applying both the cache and NUMA optimizations in the pull direction. Table 10 shows GraphIt's reduced LLC miss rate, QPI traffic, and cycles stalled.

BC. GraphIt achieves the fastest BC performance on the USAroad and Friendster graphs and has comparable performance on the other graphs. GraphIt is a bit slower than Gemini on Twitter and WebGraph because it does not support bitvector as a layout option for vertex data vectors layouts. We plan to add this in the future.

7.2 Performance of Different Schedules

Fig. 22 demonstrates the impact of traversal direction, data structures used for keeping track of active vertices, and cache optimizations. For a given algorithm, there is no single schedule that works well on all input graphs. For BFS, DensePullSparsePush with cache optimizations reduces the number of memory accesses on LiveJournal, Twitter, WebGraph, and Friendster, achieving up to 30× speedup. However, using only SparsePush can reduce the runtime overhead on USAroad as described in Section 7.1. For CC, the bitvector and cache optimizations improve locality of

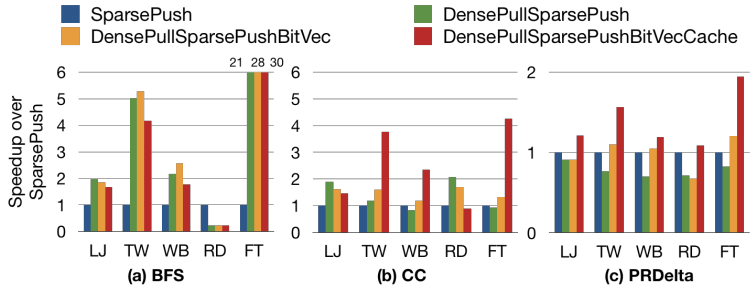

Fig. 22. Performance of different schedules for BFS, CC, and PRDelta. SparsePush and DensePullSparsePush refer to the traversal directions. BitVec refers to the dense frontier data structure. Cache refers to the cache optimization. The descriptions of these schedules can be found in Section 2.2. The full scheduling commands are shown in Table 8.

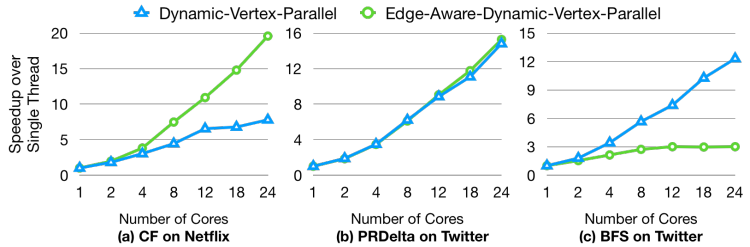

Fig. 23. Scalability of CF, PRDelta, and BFS with different schedules. Hyper-threading is disabled.

memory accesses for Twitter, WebGraph, and Friendster, but hurt the performance of LiveJournal and USAroad due to lower work-efficiency. For PRDelta, SparsePush sometimes outperforms DensePullSparsePush, but when the cache optimization is applied to the pull direction, hybrid traversal is preferred.

Fig. 23 shows that the parallelization scheme can have a major impact on scalability, and again there is no single scheme that works the best for all algorithms and inputs. For CF, the amount of work per vertex is proportional to the number of edges incident to that vertex. Consequently, the edge-aware-dynamic-vertex-parallel scheme is 2.4× faster than the dynamic-vertex-parallel approach because of better load balance. For PRDelta, the number of active vertices quickly decreases, and many of the edges do not need to be traversed. As a result, the edge-aware-dynamic-vertex-parallel scheme has a smaller impact on performance. The dynamic-vertex-parallel approach is a good candidate for BFS because not all edges incident to a vertex are traversed. Using the edge-aware-dynamic-vertex-parallel scheme for BFS ends up hurting the overall load balance. We omit the edge-parallel approach because it is consistently worse than edge-aware-dynamic-vertex-parallel due to the extra synchronization overhead.

7.3 Autotuning Performance

The autotuner found schedules that performed within 5% of the hand-tuned schedules for all of the benchmarks in under 5000 seconds. For three benchmarks, the autotuner found schedules that outperformed hand-tuned schedules by up to 10%.

7.4 Fusion of Multiple Graph Kernels

Fig. 24 demonstrates the performance improvement of kernel fusion with PageRank and Eigenvector Centrality. They have similar memory access patterns. GraphIt significantly improves the spatial locality of the memory accesses by fusing together the two kernels and the vectors they access

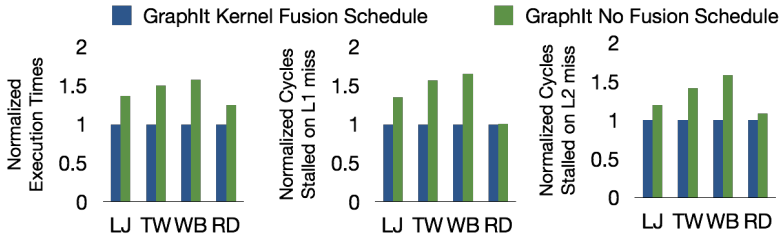

Fig. 24. Normalized Execution Time, and L1 and L2 Cache Stall Cycles with Fusion of PageRank and Eigenvector Centrality

Table 11. Optimizations adopted by various frameworks (explained in Section 2): WSVP (work-stealing vertex-parallel), WSEVP (work-stealing edge-aware vertex-parallel), SPVP (static-partitioned vertex-parallel with no work-stealing), EP (edge-parallel), BA (dense boolean array), BV (dense bitvector), AoS (Array of Structs), SoA (Struct of Arrays), SPS (SparsePush), DPS (DensePush), SP (SparsePull), DP (DensePull), SPS-DP (hybrid direction with SPS and DP depending on frontier size), DPS-SPS (hybrid with DPS and SPS), DPS-DP (hybrid with DPS and DP).

Frameworks	Traversal Directions	Dense Frontier Data Layout	Parallelization	Vertex Data Layout	Cache Opt.	NUMA Opt.	Optimization Combinations Count	Integer Params Count
GraphIt	SPS, DPS, SP, DP, SPS-DP, DPS-SPS	BA, BV	WSVP, WSEVP, SPVP, EP	AoS, SoA	Partitioned, No Partition	Partitioned, Interleaved	**100+**	3
Ligra	SPS-DP, DPS-SPS	BA	WSVP, EP	SoA	None	Interleaved	4	1
Green-Marl	DPS, DP	BA	WSVP	SoA	None	Interleaved	2	0
GraphMat	DPS, DP	BV	WSVP	AoS	None	Interleaved	2	0
Galois	SPS, DP, SPS-DP	BA	WSVP	AoS	None	Interleaved	3	1
Polymer	SPS-DP, DPS-SPS	BA	WSVP	SoA	None	Partitioned	2	0
Gemini	SPS, DP, SPS-DP	BA,BV	WSVP	SoA	None	Partitioned	6	1
GraphGrind	SPS-DP, DPS-SPS	BA	WSVP	SoA	None	Partitioned, Interleaved	4	1
Grazelle	DPS, DP, DPS-DP	BV	EP	SoA	None	Partitioned	3	1

(vertex data layout optimization). Fig. 24 shows significant reduction in cycles stalled on L1 data cache and L2 cache misses, leading to the speedups.

8 RELATED WORK

Shared-Memory Graph Processing Libraries and DSLs. Many high-performance graph frameworks and DSLs, including GraphIt, optimize their performance for shared-memory systems. Many of these frameworks support only a limited set of combinations of optimization techniques as shown in Table 11 (these optimizations are described in Section 2). *GraphIt significantly expands the space of optimizations by composing large number of effective optimizations, supporting two orders of magnitude more optimization combinations than existing frameworks.* GraphIt achieves high performance by enabling programmers to easily find the best combination of optimizations for their specific algorithm and input graph. GraphIt also finds previously unexplored combinations of optimizations to significantly outperform state-of-the-art frameworks on many algorithms.

Many shared-memory graph systems, such as Ligra [Shun and Blelloch 2013], Gunrock [Wang et al. 2016], GraphGrind [Sun et al. 2017], Polymer [Zhang et al. 2015], Gemini [Zhu et al. 2016] and Grazelle [Grossman et al. 2018], adopt the frontier-based model. Galois [Nguyen et al. 2013] also has an implementation of the model and a scheduler that makes it particularly efficient for road graphs. The frontier-based model [Shun and Blelloch 2013] operates efficiently on subsets of vertices

(frontiers) and their outgoing edges using the direction optimization [Beamer et al. 2012]. Flat data-parallel operators are used to apply functions to the frontier vertices and their neighbors with parallel direction optimizations. Existing frameworks only support up to three of the many possible directions, with little support for different parallelization schemes and frontier and vertex data layout optimizations. GraphIt significantly expands the space of optimizations by enabling combinations of data layout optimization, different direction choices, and various parallelization schemes (Table 11). GraphIt also makes programming easier by freeing the programmer from specifying low-level implementation details, such as updated vertex tracking and atomic synchronizations.

Many frameworks and techniques have been introduced to improve locality with NUMA and cache optimizations. GraphGrind, Grazelle, Gemini and Polymer all support NUMA optimizations. CSR Segmenting [Zhang et al. 2017] and cache blocking [Beamer et al. 2017; Nishtala et al. 2007] have been introduced to improve the cache performance of graph applications through graph partitioning. However, both techniques have not been integrated in a general programming model or combined with direction optimizations. GraphIt supports NUMA optimizations and integrates a simplified variant of CSR segmenting to compose cache optimizations with other optimizations.

Other shared-memory systems [Sundaram et al. 2015; Yan et al. 2017] adopt the vertex-centric model to exploit data parallelism across vertices. Programmers specify the logic that each (active) vertex executes iteratively. Frameworks [Kang et al. 2011; Lugowski et al. 2012] use sparse matrix-vector multiplication with semirings to express graph algorithms. However, both programming models cannot easily integrate direction optimization, which requires different synchronization strategies for each vertex in push and pull directions.

Green-Marl [Hong et al. 2012], Socialite [Lam et al. 2013], Abelian [Gill et al. 2018], and Empty-Headed [Aberger et al. 2016] are DSLs for shared-memory graph processing. Green-Marl provides a BFS primitive, and so programs that can be expressed with BFS invocations are relatively concise. However, for other graph programs, the programmer needs to write the loops over vertices and edges explicitly, making it hard to integrate direction optimization due to the lower level nature of the language. Socialite and EmptyHeaded provide relational query languages to express graph algorithms. The underlying data representation is in the form of tables, and due to extensive research in join optimizations, these systems perform especially well for graph algorithms that can be expressed efficiently using joins (e.g., subgraph finding). However, since these languages do not allow for explicit representation of active vertex sets, their performance on graph traversal algorithms is worse than the frontier-based frameworks [Aberger et al. 2016; Satish et al. 2014]. These DSLs also do not support composition of optimizations or expose extensive performance tuning capabilities.

A number of graph processing frameworks have been developed for GPUs (see [Shi et al. 2018] for a survey). We did not focus on GPUs in this paper as the current GPU memory capacities do not allow us to process very large graphs in-memory.

Out-of-Core Graph Processing Frameworks. There has been a significant amount of work dealing with graphs that cannot fit in memory (e.g., [Jun et al. 2018; Kyrola et al. 2012; Maass et al. 2017; Roy et al. 2013; Vora et al. 2016; Wang et al. 2017, 2015; Yan et al. 2017; Zheng et al. 2015; Zhu et al. 2015]), while GraphIt focuses on in-memory graph processing. Some of the optimizations in out-of-core systems also focus on improving locality of accesses, parallelism, and work-efficiency, but the tradeoff space for these techniques is very different when optimizing for the disk/DRAM boundary, instead of the DRAM/cache boundary. The higher disk access latency, lower memory bandwidth, and larger granularity of access lead to very different techniques [Jun et al. 2018]. When the graphs do fit in memory, out-of-core systems, such as X-Stream [Roy et al. 2013], have shown to be much slower than shared-memory frameworks [Grossman et al. 2018; Zhang et al. 2017].

Distributed Graph Processing Frameworks. Graph analytics has also been studied extensively in distributed memory systems (e.g., [Chen et al. 2015; Dathathri et al. 2018; Gonzalez et al. 2012; Low et al. 2010; McCune et al. 2015; Prabhakaran et al. 2012; Roy et al. 2015; Sakr et al. 2017; Yan et al. 2017; Zhu et al. 2016]). The tradeoff space is also different for distributed graph processing systems due to the larger network communication overhead and greater need for load balance. Techniques used by GraphIt, such as direction optimization and locality enhancing graph partitioning can also be applied in the distributed domain [Zhu et al. 2016]. These systems, when run on a single machine, generally cannot outperform shared-memory frameworks [Satish et al. 2014].

Scheduling Languages. GraphIt introduces an expressive scheduling language. Examples of existing scheduling languages include Halide [Ragan-Kelley et al. 2017], CHiLL [Chen et al. 2008], and HMPP [Romain et al. 2007]. These languages mainly focus on loop nest optimization in applications that manipulate dense arrays. Unlike these scheduling languages, the GraphIt scheduling language is designed for graph applications. It is the first scheduling language designed to address the challenges of graph applications, graph data structures, and graph optimizations. It allows the programmer to perform data layout transformations (which is not possible in Halide), and allows full separation between the algorithm and the schedule (which is not possible in HMPP). Unlike CHiLL, which was designed mainly for the application of affine transformations on loop nests, the GraphIt scheduling language supports a large set of non-affine transformations, which are the main type of optimizations in the context of graph applications.

Program Synthesis. Program synthesis techniques have been explored in the context of graph algorithms [Prountzos et al. 2012, 2015], which allow many different implementations of an application to be generated. However, little control is provided to compose together different optimizations. GraphIt enables programmers to apply their knowledge to find profitable combinations of optimizations. GraphIt also supports a much wider range of optimizations.

Physical Simulation DSLs. GraphIt is heavily influenced by DSLs for physical simulations, including Simit [Kjolstad et al. 2016] and Liszt [DeVito et al. 2011]. However, Simit and Liszt do not support efficient filtering on vertices and edges, and do not have a scheduling language.

9 CONCLUSION

We have described GraphIt, a novel DSL for graph processing that generates fast implementations for algorithms with different performance characteristics running on graphs with varying sizes and structures. GraphIt separates algorithm specifications from performance optimizations. The algorithm language simplifies expressing the algorithms. We formulate graph optimizations as tradeoffs among locality, parallelism, and work-efficiency. The scheduling language enables programmers to easily search through the complicated tradeoff space. We introduce the *graph iteration space* to model, compose, and ensure the validity of the edge traversal optimizations. The separation of algorithm and schedule, and the correctness guarantee of edge traversal optimizations enabled us to build an autotuner on top of GraphIt. Our experiments show that GraphIt is up to 4.8× faster than state-of-the-art graph frameworks. Future work includes extending the compiler to support more optimizations and hardware platforms (e.g., GPU and distributed-memory).

ACKNOWLEDGMENTS

We thank Tyler Denniston, Vladimir Kiriansky, Jure Leskovec, Michael W. Mahoney, and the reviewers for their helpful feedback and suggestions. This research was supported by Toyota Research Institute, DoE Exascale award #DE-SC0008923, DARPA SDH Award #HR0011-18-3-0007, DARPA D3M Award #FA8750-17-2-0126, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, and DOE Early Career Award #DE-SC0018947.

REFERENCES

Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *International Conference on Management of Data (SIGMOD '16)*. 431–446.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*.

Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. Article 12, 12:1–12:10 pages.

Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *IEEE International Symposium on Workload Characterization (IISWC)*. 56–65.

Scott Beamer, Krste Asanovic, and David Patterson. 2017. Reducing Pagerank Communication via Propagation Blocking. In *IEEE International Parallel and Distributed Processing Symposium*. 820–831.

James Bennett, Stan Lanning, and Netflix Netflix. 2007. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*.

Maciej Besta, Michal Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. 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 (HPDC '17)*. 93–104.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report.

Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Article 1, 1:1–1:15 pages.

Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

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

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

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. Article 9, 12 pages.

Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. 1775–1784.

Zhisong Fu, Zhengwei Wu, Houyu Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. 2017. GeaBase: A High-Performance Distributed Graph Database for Industry-Scale Applications. In *International Conference on Advanced Cloud and Big Data (CBD)*. 170–175.

Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. 2018. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. 249–264. https://doi.org/10.1007/978-3-319-96983-1_18

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30.

Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-based Graph Processing Performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 246–260.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *SIGARCH Comput. Archit. News* 40, 1 (March 2012), 349–362.

Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Accelerated Flash Storage for External Graph Analytics. In *International Symposium on Computer Architecture*.

U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2011. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.* 27, 2 (2011).

Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing Indirect Memory References with Milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. 299–312.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (March 2016), 20:1–20:21 pages.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. 591–600.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 31–46.

Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *ICDE*. 278–289.

Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

Boduo Li, Sandeep Tata, and Yannis Sismanis. 2013. Sparkler: Supporting Large-scale Matrix Factorization. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. 625–636.

Zhiyuan Li, Pen-Chung Yew, and Chuag-Qi Zhu. 1989. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 3rd international conference on Supercomputing*. 215–224.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework For Parallel Machine Learning.. In *UAI*. 340–349.

Adam Lugowski, David Alber, Aydın Buluç, John Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. 2012. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *SDM*.

Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 527–543.

Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *USENIX Annual Technical Conference*.

Dror E Maydan, John L Hennessy, and Monica S Lam. 1991. Efficient and exact data dependence analysis. In *ACM SIGPLAN Notices*, Vol. 26. 1–14.

Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2, Article 25 (Oct. 2015), 25:1–25:39 pages.

Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. [n. d.]. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph. ([n. d.]).

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 456–471.

Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. 2007. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing* 18, 3 (01 May 2007), 297–311.

David A Padua and Michael J Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab.

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. 2012. Managing Large Graphs on Multi-cores with Graph Awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: a system for synthesizing concurrent graph programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 375–394.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2015. Synthesizing parallel graph programs via automated planning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 533–544.

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Commun. ACM* 61, 1 (Dec. 2017), 106–115.

Dolbeau Romain, Stephane Bihan, and Francois Bodin. 2007. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*.

Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 410–424.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 472–488.

Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2017. *Large-Scale Graph Processing Using Apache Giraph* (1st ed.). Springer Publishing Company, Incorporated.

Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 979–990.

Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1281–1292.

Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6, Article 81 (Jan. 2018), 81:1–81:35 pages.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. 135–146.

Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. Article 16, 16:1–16:10 pages.

Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225.

Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference*. 507–522.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 389–404.

Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *USENIX Annual Technical Conference*. 387–401.

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (Feb. 2016), 12 pages.

Michael E Wolf and Monica S Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (1991), 452–471.

Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

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

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

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 45–58.

Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 301–316.

Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. 375–386.