One Graph, Many Sources: Automating Vectorization Safely

Wenyue Zhao, Yang Cao, Peter Buneman
The University of Edinburgh, UK
{wenyue.zhao, yang.cao, peter.buneman}@ed.ac.uk

Jia Li, Nikos Ntarmos
Edinburgh Research Center, Huawei, UK
{jia.li3, nikos.ntarmos}@huawei.com

ABSTRACT

Many graph workloads repeatedly run the same traversal or iterative computation from many different source vertices. Multi-instance execution can share work across sources and, if implemented carefully, exploit SIMD by packing per-source state into vectors. However, in vertex-centric graph processing systems, naive "vectorize everything" transformations can be wrong: a vertex may be active for only a subset of sources in a round, yet unguarded SIMD updates implicitly advance all lanes, leading to incorrect answers.

We present AutoMI, a source-to-source compilation framework that automatically converts a single-instance program into a provably correct SIMD-vectorized multi-instance program runnable on existing vertex-centric engines. AutoMI uses a per-message bit-vector `track` to recover per-source activation and generate masked SIMD updates. It further provides a TrackFree optimization, guided by an algebraic idempotence characterization, that safely drops `track` and produces streamlined code when over-activation is harmless.

On six real graphs (up to billions of edges) and up to 256 sources, AutoMI achieves 9.6 to 29.5× speedup over parallelized serial evaluation, 7.1 to 26.4× over batch processing, and 2.6 to 4.6× over prior handcrafted multi-instance code.

1. INTRODUCTION

Modern graph workloads rarely run a graph algorithm just once. In social analysis, web search and recommendation, it is common to execute the same traversal or iterative computation many times on the same graph but with different source vertices. Specifically, given a graph $G = (V, E)$ and a source-parametric query $Q(s)$, the task is to compute $Q(s_1), \ldots, Q(s_k)$ for a set of sources $s_1, \ldots, s_k$. Examples include classic traversal queries like reachability, breadth-first search (BFS), single-source shortest paths (SSSP), as well as advanced graph analytics like sparse matrix-vector multiplication and graph learning [23], where computations are recast as iterative vertex updates on an induced graph.

Multi-instance algorithms. A straightforward approach is *serial evaluation* that runs a standard graph algorithm once per source, in parallel. While easy to implement, this repeats substantial work, both computation and communication (distributed setting), and fails to exploit a key observation: different sources often traverse overlapping regions of the graph. This has motivated *multi-instance* graph algorithms [33, 32, 12, 21, 29, 10, 30, 18, 35, 38, 22, 42], which interleave multiple instances to share traversal and computation. When instances execute similar operations on the same vertices, multi-instance execution opens door to SIMD vectorization acceleration by packing per-source state into vectors and applying the same instruction across instances.

Unfortunately, implementing multi-instance algorithms and obtaining these benefits can be notoriously challenging. Even for simple algorithms such as BFS, it takes substantial and delicate effort to ensure that the benefits of multi-instance computation outweigh the overhead of aligning the computations across sources. For instance, a multi-instance version of BFS requires approximately 600 lines of core code to be effective despite the simple algorithmic logic [35], without even applying SIMD vectorization yet [22]. A transformation that "packs per-source state into vectors and applies SIMD throughout" can be *incorrect*, even in vertex-centric frameworks [25, 23, 27] dedicated to parallel graph computations.

The core challenge is that, during graph traversal a vertex can be active for some sources but not others in the same round. Naive SIMD vectorization implicitly assumes that all lanes are active whenever a vertex function fires, which can corrupt per-source state (*e.g.,* marking a vertex as visited for sources that have not reached it yet) and lead to wrong answers. Preventing this would require extra work that carefully aligns the computations and progresses across sources, which, if not done carefully, may bring in overhead that outweigh the benefits of multi-instance implementation.

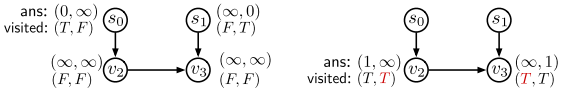
Automating multi-instance computations. In response to the challenge of developing effective multi-instance algorithms, we present AutoMI, a compilation framework that automatically converts a standard single-instance vertex-centric algorithm [2, 25, 17, 23, 15, 13] written in the GAS (Gather-Apply-Scatter) programming model [15, 13, 27, 36, 16, 24, 39, 11] into a *vectorized* multi-instance GAS program that runs directly on existing distributed engines. AutoMI ensures correctness by introducing a per-message bit-vector, `track`, which records *which sources* are logically responsible for a vertex activation. During code generation, AutoMI au-

ALGORITHM 1: Single-instance Integer BFS (iBFS)

```
1  #define vertex_data { <int> ans, <bool> visited } ;        // vertex properties
2  #define msg { <int> ans } ;                                // define message struct
3  Function Gather(vertex u, edge (u, v), vertex v)          // invoked at v (center vertex)
4      return msg                                             // sent from u by Scatter in previous round
5  Function Gather_Acc(vertex v, msg_acc, msg)               // center vertex: v
6      msg_acc.ans ← Min (msg_acc.ans, msg.ans) ;             // accumulate and aggregate msg in msg_acc; initially, msg_acc.ans is 0 if v is source and +∞ otherwise
7      return msg_acc;
8  Function Apply(vertex v, msg_acc)                         // center vertex: v
9      if NOT v.visited then
10         v.ans ← msg_acc.ans;
11         v.visited ← True;
12         changed ← True ;                                   // mark changed as True
13 Function Scatter(vertex v, msg_acc)                       // center vertex: v
14     if changed then                                        // check if changed is True
15         msg.ans ← v.ans + 1 ;
16         Signal (msg) ;                                     // activate and send msg to outward neighbors
```

ALGORITHM 2: Multi-instance Integer BFS

```
1  #define vertex_data { array <int> ans, bitvec <bool> visited } ;
2  #define msg { array <int> ans, bitvec <bool> track } ;
3  Function Gather(vertex u, edge (u, v), vertex v)
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.track ← simd_Or (msg_acc.track, msg.track) ;
7      msg_acc.ans ← simd_mask_Min (msg.track, msg_acc.ans, msg.ans);
       return msg_acc;
8  Function Apply(vertex v, msg_acc)
9      mask ← simd_mask_Neg (msg_acc.track, v.visited) ;
10     v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans) ;
11     v.visited ← simd_mask_Set (mask, v.visited, 1) ; // 1: constant True vector
12     changed ← simd_mask_Set (mask, changed, 1) ;
13 Function Scatter(vertex v, msg_acc)
14     mask ← simd_mask_Set (msg_acc.track, mask, changed) ;
15     msg.ans ← simd_mask_Add (mask, v.ans, 1);
16     if NOT simd_All_Zero (mask) then
17         msg.track ← simd_Set (msg.track, mask) ;
18         Signal (msg) ;
```



(a) before round 1 (initial state) **(b)** after round 2

Figure 1: Data graph for Integer BFS (sources: s_0 and s_1)

tomatically turns `track` into SIMD masks that guard vectorized updates, so each lane evolves exactly as it would under an independent single-source execution.

Over-vectorization and `TrackFree`. Unlike generic programs where incorrectly activated lanes are always harmful as they lead to incorrect answers, there are certain graph algorithms for which one can safely *over-vectorize*: we simply activate vertex functions for all sources even if not all of them have reached the vertex, and we assure that even if we "over-activate" vertices, the algorithm converges to the same fixed point as the correct multi-instance algorithm would do.

AutoMI captures this case via a `TrackFree` optimization that removes `track` entirely and generates streamlined, more efficient SIMD code. To decide when `TrackFree` is safe, we extend linear algebra to allow users to abstract algorithmic logic of the Apply and Scatter functions of their GAS programs in the extended algebra. We then develop a formal characterization of track-free vectorization of single-instance GAS algorithms by reducing to two *algebraic idempotence* properties of their algebra abstraction, allowing users to decide when to enable track-free vectorization in AutoMI.

On billion-edge real-life graphs, AutoMI-generated programs deliver substantial speedups (up to roughly 10×–30× over serial evaluation, and up to 2.6×–4.6× over handcrafted highly optimized multi-instance implementations)

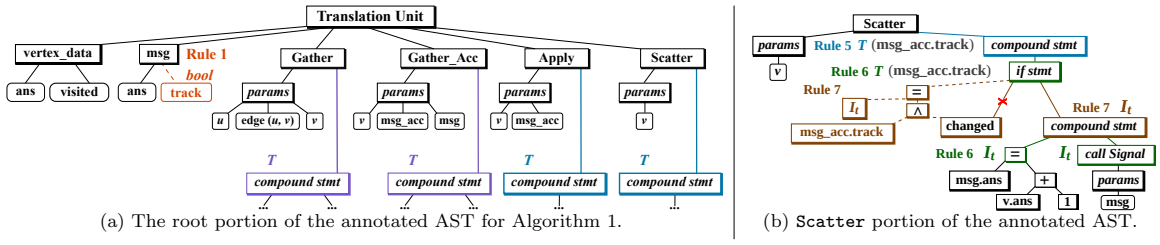
Takeaway. Multi-instance algorithms can be highly efficient, but implementing them correctly and efficiently, especially when exploiting SIMD, is notoriously tricky. AutoMI makes "multi-instance + SIMD" practical without requiring developers to redesign algorithms or adopt specialized programming models. It does so via a source-to-source compilation framework that generates provably correct multi-instance GAS programs, plus a principled `TrackFree` optimization that identifies maximum performance benefits from SIMD vectorization without impairing correctness guarantees.

2. WHY NAIVE VECTORIZATION FAILS

Graph queries and computations. We consider a (typically large) directed graph $G = (V, E)$, possibly with edge properties such as weights. A query instance is parameterized by a source vertex $s \in V$ and returns, for each vertex $v \in V$, an answer value determined by the query semantics. Classic traversal instances include reachability (Boolean BFS), levels (Integer BFS), and single-source shortest paths (SSSP), where information is propagated along edges from s.

Beyond traversals, AutoMI also targets common graph analytics kernels that can be expressed in the same vertex-centric style. We list some examples below. (a) Sparse Matrix-Vector multiplication (SpMV), which computes an output vector $y = Ax$ where A is (a weighted variant of) the adjacency matrix of G and x is an input vector with one entry per vertex (often source-dependent in personalized settings). Since A is $|V| \times |V|$, the result y is also a length-$|V|$ vector, *i.e.,* it returns one value $y[v]$ per vertex, obtained by aggregating contributions from neighbors, *e.g.,* $y[v] = \sum_{(u,v)\in E} w(u, v)\, x[u]$. (b) This is similar for Personalized PageRank (PPR), which iteratively applies SpMV-style propagation together with a source-specific personalization/restart distribution. (c) Collaborative filtering (CF), which iteratively updates per-vertex latent factors by stochastic gradient descent driven by edge interactions, can also be recast similarly. These problems are not always phrased as graph traversals, but they induce vertex-centric iterative computations and can be instantiated for many sources or personalization seeds in the same way as BFS and SSSP.

Vertex-centric programs. AutoMI targets vertex-centric graph systems (*e.g.,* Pregel [25], PowerGraph [15] and PowerLyra [13]) that run a user program "at each vertex" in synchronous rounds (*a.k.a.* supersteps) [25, 15, 13]. In the GAS (Gather-Apply-Scatter) model [15, 13, 26], an active vertex (i) aggregates messages from its neighbors (Gather), (ii) updates its local vertex properties (Apply), and (iii) decides whether to send messages and activate neighbors for the next round



(a) The root portion of the annotated AST for Algorithm 1.

(b) `Scatter` portion of the annotated AST.

Figure 2: The annotated AST of Algorithm 1 (dashed lines represent added edges during annotation)

(Scatter). Execution terminates when no vertex is active.

GAS program for Integer BFS. Algorithm 1 gives a standard Integer BFS in the GAS model. Given a graph G and a source vertex s, it computes for each vertex v its BFS depth from s. The program maintains two vertex properties: v.`ans` (the current depth) and v.visited (whether v has been visited); messages carry candidate depths between neighbors.

A GAS engine executes this vertex program in synchronous rounds on *active* vertices (initially only s; thereafter, vertices signaled in the previous round). Each round consists of:

(1) *Gather.* Each active v pulls messages from its in-neighbors and `Gather_Acc` aggregates them into an accumulated message `msg_acc` (here, the minimum depth; lines 3-7).

(2) *Apply.* This is the main logic of vertex programs. For BFS, it implements the "first-visit" rule: if v.visited is false, commit `msg_acc.ans` to v.`ans`, set v.visited and `changed` to true, where `changed` is a reserved per-vertex temporary flag reset to *False* at the start of each superstep (lines 8-12).

(3) *Scatter.* Finally, if `changed` is true, `Scatter` sends a message encoding "v.`ans` plus 1" to v's out-neighbors, thereby activating them next round (lines 13-16).

A GAS system would invoke Algorithm 1 as the vertex function of BFS at all active vertices, round by round, until no vertex is signaled (active); it then returns the ans property of all vertices. By selectively activating vertices, systems need only invoke programs for vertices that require computation instead of blindly running at all vertices, making them practical and scalable [25, 27, 37]. Moreover, this allows the core GAS logic to focus on the "real" actions needed for every active vertex, yielding simpler algorithm implementations that follow the "thinking-as-a-vertex" programming logic, inherently more natural on graphs [27].

Multi-instance GAS. A natural way to accelerate multi-source workloads is to evaluate many query instances together, rather than running a single-instance GAS program k times (serial evaluation). In a multi-instance setting, we want one GAS run to compute $Q(s_1), \ldots, Q(s_k)$ by interleaving the k instances so that they can share traversal, computation and communication, and so that the same vertex operation can be applied to many sources in parallel.

While selective vertex activation is crucial for the simplicity and scalability of vertex-centric programs, as we will see below it is also what makes vectorizing GAS programs for correct multi-instance execution subtle: in a given round, a scheduled vertex may be active for only a subset of sources.

Populate-and-vectorize. The most straightforward way to obtain a multi-instance GAS program is to *populate* the single-instance one: lift each vertex property (*e.g.,* ans, visited) and each message field from a scalar to a length-k vector, where the i-th lane stores the value for source s_i. Then, replace scalar operations over these properties with their lane-wise vector counterparts. This "populate-and-vectorize" transformation is appealing because it preserves the structure of the original GAS code and, with an appropriate memory layout, directly enables SIMD execution: the same instruction updates multiple lanes (sources) at once.

Why does SIMD vectorization not work? The populate-and-vectorize transformation is attractive but *not* correct in general. Vertex-centric engines schedule computation at the *vertex* granularity: in round t, a vertex v executes if it is signaled by *any* instance, but the engine does not record which sources are responsible for the activation. Hence, when a vectorized vertex function fires at v, only a subset of SIMD lanes may correspond to sources that should make progress at v in this round. Updating the remaining lanes can corrupt per-source state and change later control flow.

Vectorized BFS (counterexample). Consider Integer BFS. If we drop the shaded track-maintenance lines from Algorithm 2, the remaining code is precisely a streamlined vectorization: vertex properties are lifted to vectors and Apply/Scatter execute lane-wise (with masks derived only from local predicates such as "not visited"). Figure 1 shows the failure with two sources s_0 and s_1. After round 1, v_2 is reached only from s_0 and v_3 only from s_1. In round 2 the engine invokes Apply at both vertices, but the streamlined program has no mask that captures *which source activated the vertex*. Since visited is initially false for both lanes, it sets visited for *both* s_0 and s_1 at both v_2 and v_3. The lane for s_0 is thus incorrectly marked as having visited v_3, and the later correct update of v_3.`ans` to $(2, 1)$ is suppressed.

We remark that streamlined vectorization is inherently algorithm dependent: for example, we show that Bellman-Ford (Algorithm 4, Section 5) can also compute BFS depths on unweighted graphs and remains correct under streamlined vectorization [41], while Integer BFS does not. For some workloads, streamlined vectorization may be elusive altogether because their updates are not monotone under over-activation (*e.g.,* we are not aware of any streamlined multi-instance collaborative filtering). These subtleties motivate AutoMI: it automatically converts single-instance GAS programs into correct vectorized multi-instance ones (Section 3) and provides an algebraic characterization to decide when streamlined vectorization can be enabled safely (Section 4).

3. VECTORIZING GRAPH ALGORITHMS

To deal with the subtlety of writing correct and efficient multi-instance algorithms, we develop AutoMI, a framework

Rule 1 Message Structure		
`#define msg {...}`	⇒	`#define msg {...` mark(T) `<bool> track}`
Rules 2-5 `Gather`, `Gather_Acc`, `Apply`, `Scatter` Functions		
Function Name (*args ...*) *stmt*	⇒	Function Name (*args ...*) mark(T) *stmt*
Rule 6 Compound Statement		
mark(ℓ) {$stmt_1 ... stmt_n$} // ℓ: 'T' or new var (rule 7)	⇒	mark(ℓ) $stmt_1$ ... mark(ℓ) $stmt_n$
Rule 7 IF Selection Statement		/* I_t: a distinct variable*/
mark(ℓ) **if** E **then** $stmt_t$ **else** $stmt_f$	⇒	$I_t \leftarrow \ell \wedge E$ **if** I_t **then** mark(I_t) $stmt_t$ $I_f \leftarrow \ell \wedge \neg I_t$ **else** mark(I_f) $stmt_f$

Table 1: Annotation rules: *each line on* RHS *of the rules corresponds to a node of the annotated AST.*

Rule 1 Structure Definition (using `msg` as example)		
`Codegen ( #define msg {...` `<bool> track} )`	⇒	`#define msg {...` `bitvec<bool> track }`
Rule 2 `Gather_Acc` Function Definition		

`Codegen (`
`Gather_Acc(`*args ...*`)`
mark(T) *stmt*
)
	⇒	`Gather_Acc(`*args ...*`)`
		`msg_acc.track ← simd_Or(`
		`msg_acc.track, msg.track)`
		`Codegen(` mark(T) *stmt* `)`

Rule 3 `Gather`, `Apply`, `Scatter` Function Definition		
`Codegen (` Function Name (*args ...*) mark(T) *stmt*)	⇒	Function Name (*args ...*) `Codegen(` mark(T) *stmt*)
Rule 4 Signal Function Call		

`Codegen (`
mark(ℓ) `Signal(msg))`
	⇒	**if** $\ell \neq$ **0 then** /*NOT simd_All_Zero (ℓ)*/
		`msg.track ← simd_Set(msg.track,` ℓ`)`
		`Signal(msg)`

Rule 5 Assignment Statement		

`Codegen (` mark(ℓ) *var* ← *e* `)`
`Codegen (` mark(ℓ) *var* ← `Op(`e_1`) )`
`Codegen (` mark(ℓ) *var* ← e_1 `Op` e_2 `)`
	⇒	*var* ← `simd_mask_Set(`ℓ, var, e`)`
		var ← `simd_mask_Op(`ℓ, e_1`)`
		var ← `simd_mask_Op(`ℓ, e_1, e_2`)`

Rule 6 Compound Statement		
`Codegen (` mark(ℓ) $stmt_1$ ... mark(ℓ) $stmt_n$)	⇒	{ `Codegen(` mark(ℓ) $stmt_1$) ... `Codegen(` mark(ℓ) $stmt_n$) }
Rule 7 IF Selection Statement		

`Codegen (`
$I_t \leftarrow \ell \wedge E$
if I_t **then** mark(I_t) $stmt_t$
$I_f \leftarrow \ell \wedge \neg I_t$
else mark(I_f) $stmt_f$)
	⇒	`Codegen(` $I_t \leftarrow \ell \wedge E$ `)`
		`Codegen(` mark(I_t) $stmt_t$ `)`
		`Codegen(` $I_f \leftarrow \ell \wedge \neg I_t$ `)`
		`Codegen(` mark(I_f) $stmt_f$ `)`

Table 2: Code-generation rules: LHS *denotes a node of the annotated AST and* RHS *is the generated code fragment.*

for converting a given single-instance GAS program (*e.g.,* Algorithm 1) into a provably correct multi-instance program (*e.g.,* Algorithm 2) that directly runs on any GAS system.

Concretely, given a single-instance GAS program $\mathbf{A}$ and a set of k sources $s_1, \ldots, s_k$, AutoMI produces a multi-instance program $\mathbf{A}_m$ that computes $Q(s_1), \ldots, Q(s_k)$ in one run. The core challenge is to (a) deduce, at runtime, which sources are "visiting" a vertex v when it is activated, and (b) use this information to guard SIMD applications so that each lane evolves exactly as in an independent execution of $\mathbf{A}$.

AutoMI introduces a new per-message bit-vector field track. Intuitively, `msg.track` records which sources are responsible for the activation carried by that message. As messages are aggregated in Gather, AutoMI maintains an accumulated mask `msg_acc.track` at each active vertex; `msg_acc.track` is then used as the SIMD mask that guards `Apply`/`Scatter` updates. This directly addresses the failure mode of the "populate-and-vectorize" vectorization in Section 2: even though the engine schedules at the vertex level, track reconstructs lane-level activation information needed for correctness.

AutoMI: single-instance ⇒ multi-instance. AutoMI performs the source-to-source conversion in two steps.

- AST annotation: AutoMI first parses the input GAS program into an abstract syntax tree (AST) and annotates statements with labels that will later be interpreted as SIMD masks (Table 1).
- Vectorized code generation: AutoMI traverses the annotated AST and generates a vectorized multi-instance GAS program by (i) populating data structures and (ii) emitting SIMD operations guarded by masks derived from annotations (Table 2).

Step (1): AST annotation (Table 1). AutoMI first obtains an AST of the input GAS program via a compiler front-end, and then alters and annotates it via rules in Table 1:

(a) Introduce track. Rule 1 extends the message structure `msg` with a Boolean field track.

(b) Annotate. Rules 2-6 propagate a default label T through the statements of GAS functions. Intuitively, T means "execute under the current activation mask"; its exact interpretation depends on the phase, as we will see below.

(c) Alter AST & refine labels. Rule 7 rewrites an **if** statement under label ℓ into two per-branch masks $I_t \leftarrow \ell \wedge E$ and $I_f \leftarrow \ell \wedge \neg I_t$, and annotates then/else subtrees with I_t and I_f, respectively. This makes control flow lane-aware: each branch will execute only for lanes that satisfy its predicate.

Example 1: Figure 2 illustrates AutoMI's two-step conversion on Algorithm 1. In the annotation step, AutoMI (i) extends `msg` with the track field (Rule 1) and (ii) propagates the default label T through the four GAS functions (Fig. 2a). In `Scatter` (Fig. 2b), it rewrites each **if** under label ℓ into per-lane masks $I_t \leftarrow \ell \wedge E$ and $I_f \leftarrow \ell \wedge \neg I_t$ (Rule 7). □

Step (2): vectorized code generation (Table 2). Given the annotated AST, AutoMI generates a multi-instance GAS program by a depth-first traversal. Its code generator `Codegen` interprets annotations as SIMD masks via rules of Table 2.

(a) Structure populating. For k sources, each scalar vertex property (*e.g.,* `ans`) is lifted to a length-k array storing per-source values; similarly for message fields. In addition, `msg` is extended with track now represented as a bit-vector (Rule 1).

(b) Propagating track *through GAS.* In `Gather_Acc`, AutoMI accumulates the incoming `msg.track` into `msg_acc.track` via bitwise OR in Rule 2. In `Scatter`, when a vertex signals neighbors, AutoMI sets `msg.track` to the current mask and only sends the message if at least one lane is active (Rule 4). These ensure that for each vertex v and round, `msg_acc.track` identifies exactly lanes that logically activate v.

(c) Track-guarded SIMD updates. Assignments and operations on vertex properties are compiled into SIMD operations guarded by the current mask label (Rules 5-7). It interprets T as an all-ones mask in `Gather`/`Gather_Acc` (since the phase is purely message transport/aggregation), and as `msg_acc.track` in `Apply`/`Scatter` (so updates occur only for active lanes). When Rule 7 introduces a branch mask vari-

ALGORITHM 3: Multi-instance Boolean BFS (bBFS)

```
1  #define vertex_data { bitvec <bool> ans } ;
2  #define msg { bitvec <bool> ans } ;

3  Function Gather(vertex u, edge (u, v), vertex v)
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.ans ← simd_Or (msg_acc.ans, msg.ans) ;
7      return msg_acc ;

8  Function Apply(vertex v, msg_acc)
9      changed ← simd_AndNot (v.ans, msg_acc.ans) ;      // newly
       reached lanes: ¬v.ans ∧ msg_acc.ans
10     v.ans ← simd_Or (v.ans, msg_acc.ans) ;

   Function Scatter(vertex v)
11     if NOT simd_All_Zero (changed) then // if changed is True for
         some source
12         msg.ans ← simd_Set (msg.ans, v.ans) ;
13         Signal (msg) ;
```

able (*e.g.,* I_t), `Codegen` uses it directly as the mask for that branch, enabling correct per-lane control flow.

Example 2: Continuing Example 1, in the code-generation step, these labels are interpreted as SIMD masks: T becomes **1** in `Gather`/`Gather_Acc` and `msg_acc`.track in `Apply`/`Scatter`, and signaling sets `msg`.track to the current mask. This yields Algorithm 2, where updates to ans and visited apply only to sources that actually activate the vertex in that round. □

Correctness. The original full paper [41] proves that AutoMI is semantics preserving: for any graph and any set of sources, the generated program $\mathbf{A}_m$ produces exactly the per-source results of running $\mathbf{A}$ independently for each source. Intuitively, track ensures that a lane is updated at a vertex if and only if that source would have activated the vertex in a single-instance execution; the mask refinement at control flow further ensures that each lane follows the same branch decisions as in the original program.

Remark. Our rules presented here are for loop-free GAS functions. This matches most vertex programs in practice, as for and while loops are rarely used in GAS programs since they are essentially local functions running concurrently on each vertex. We remark that one can extend its coverage by adding rules to support loop constructs *e.g.,* for and while.

4. DROP MASKS (AND GO FASTER)?

Although track-guarded vectorization (Section 3) is always correct, it is not free: track enlarges messages, adds aggregation work in `Gather_Acc`, and turns many SIMD operations into masked SIMD. For some algorithms, this overhead is avoidable: even if we "over-activate" a vertex for sources that have not reached it yet, the extra executions do not change the final fixed point. We call such programs `TrackFree`.

`TrackFree` by example: Boolean BFS. Algorithm 3 is a streamlined multi-instance Boolean BFS: each vertex stores a bitvector ans encoding reachability from k sources, aggregates neighbor reachability via $\vee$, and signals only when some lane changes. Notably, the program has *no* track field and performs no per-lane activation masking. It is nevertheless correct as reachability is monotone: executing `Apply` on a lane "too early" cannot turn a reachable vertex into unreachable, and repeated applications converge to the same fixed point.

What does it mean to be `TrackFree`? Consider a single-instance GAS program $\mathbf{A}$. Let $\mathbf{A}_m$ be the general multi-instance program that AutoMI generates in Section 3, which maintains track and guards SIMD updates. Let $\mathbf{A}_m^{\mathsf{TF}}$ be the streamlined version that drops track, derived from $\mathbf{A}$ via the populate-and-vectorize transformation described in Section 2. We say $\mathbf{A}$ is `TrackFree` if, for any graph G and any set of sources, $\mathbf{A}_m^{\mathsf{TF}}$ returns exactly the same per-source answers as $\mathbf{A}_m$ (which is equivalent to running $\mathbf{A}$ independently for each source). When $\mathbf{A}$ is `TrackFree`, AutoMI can safely drop masks and generate simpler and faster vectorized code.

Determining whether $\mathbf{A}$ is `TrackFree` is subtle: Section 2 shows that Integer BFS fails under streamlined vectorization, while other formulations of the same task (e.g., Bellman-Ford on unweighted graphs) do not. We therefore develop a principled *sufficient* condition of `TrackFree` based on an abstract representation of the program, which allows us to decide `TrackFree` in a unified way rather than reasoning about all possible activation patterns case by case for individual graph algorithms and their GAS programs.

An algebraic characterization. The key difficulty in deciding `TrackFree` is that correctness depends on subtle interactions between (i) the per-round `Apply` update rule and (ii) the engine's activation schedule. Rather than reasoning about all possible interleavings case-by-case, AutoMI uses an algebraic abstraction of the program semantics. At a high level, we recast the one-round behavior of $\mathbf{A}$ into an abstract `Apply` update function $e_{\mathbf{A}}$ (parameterized by the accumulated message λ). TrackFree vectorization corresponds to executing `Apply` on *extra* lanes (either too early or when they are actually inactive). The characterization identifies when such extra executions are harmless by reducing to two idempotence conditions on $e_{\mathbf{A}}$ plus a mild regularity on `Scatter`.

Recasting one GAS round. To reason about when masks can be dropped, AutoMI abstracts a GAS program $\mathbf{A}$ by its *per-round* behavior, separating (i) how vertex state is updated in Apply from (ii) how vertices are activated in Scatter.

Assume $|V| = n$ and that $\mathbf{A}$ maintains p vertex properties. We view each property as a length-n column vector: for $j \in [1, p]$, $x_j[v]$ is the value of property j at vertex v, and we write the whole state as $\mathbf{x} = (x_1, \ldots, x_p)$. In one superstep, `Gather`/`Gather_Acc` compute an *accumulated message* for each vertex by aggregating information from neighbors. We denote this accumulated message (or tuple of message fields) by λ. In many graph kernels, λ is expressible using the adjacency matrix A of G, *e.g.,* a matrix-vector product such as $\lambda = A^T \cdot x$ under a suitable semiring; for simplicity, here we treat λ abstractly as "the aggregated input to `Apply`".

Fix an accumulated message value λ. The `Apply` phase deterministically computes the next vertex state from the current one, using only local state and λ. We denote by $e_{\mathbf{A}}(\cdot)\langle\lambda\rangle$ the (vector-valued) function induced by `Apply` that maps the current state $\mathbf{x}$ to the state after one round of `Apply` execution when the accumulated message is λ:

$$\mathbf{x}' \;=\; e_{\mathbf{A}}(\mathbf{x})\langle\lambda\rangle.$$

Similar to λ, $e_{\mathbf{A}}$ is expressible in matrix languages with a suitable semiring of choice (see the full paper [41] for details). The angle brackets emphasize that λ is treated as a parameter (held fixed), which will be used by the characterization.

Two idempotence conditions. Below we characterize `TrackFree` of $\mathbf{A}$ by relating simple properties of $e_{\mathbf{A}}$ to the correct-

ness of dropping `track` in the vectorized program.

Specifically, `TrackFree` corresponds to executing some `Apply` calls that are redundant (too early, too often, or for lanes that are not truly active). These extra calls are harmless if `Apply` is idempotent in two complementary senses.

(I1) *Active idempotence.* For any fixed accumulated message λ, applying the same update twice has no additional effect:

$$e_{\mathbf{A}}(\mathbf{x})\langle\lambda\rangle \;=\; e_{\mathbf{A}}\big(e_{\mathbf{A}}(\mathbf{x})\langle\lambda\rangle\big)\langle\lambda\rangle.$$

Intuitively, this condition means that if a vertex is scheduled multiple times under an unchanged incoming message, the second (and later) applications do not further change state.

(I2) *Inactive idempotence.* Let α be the "inactive" message induced by initialization for a vertex that has not been reached by any source (*e.g.,* $+\infty$ for shortest paths, *False* for reachability). Then executing `Apply` with α must be a no-op:

$$\mathbf{x} \;=\; e_{\mathbf{A}}(\mathbf{x})\langle\alpha\rangle.$$

This captures the key requirement for over-activation vertices during GAS rounds: running `Apply` for lanes that are not supposed to be active should not change their state.

Regular signaling. We say that `Scatter` is *regular* if either (a) a vertex always signals neighbors, or (b) it signals only when its state changes in this round (`changed` is *True*). These are common patterns in GAS traversals and iterative kernels, and they allow the algebraic reasoning to connect "who is active" with "who receives a nontrivial message."

Theorem 1: *For a GAS program* $\mathbf{A}$*, if its* `Apply` *abstraction* $e_{\mathbf{A}}$ *is both active and inactive idempotent, and its* `Scatter` *is regular, then* $\mathbf{A}$ *is* `TrackFree`*.* □

Applications through examples. Theorem 1 becomes intuitive once we instantiate $e_{\mathbf{A}}$ for familiar kernels. For Boolean BFS, the only state is reachability x, the accumulated message λ is the disjunction of neighbors' reachability, and `Apply` is $e_{\mathbf{A}}(x)\langle\lambda\rangle = x \vee \lambda$. Condition **(I1)** holds as $(x\vee\lambda)\vee\lambda = x\vee\lambda$, and **(I2)** holds because the inactive message is $\alpha = \mathit{False}$ and $x \vee \mathit{False} = x$. Moreover, `Scatter` is regular since it signals only when `changed` is true (Algorithm 3).

For Bellman-Ford/SSSP, `Apply` is $e_{\mathbf{A}}(x)\langle\lambda\rangle = \min(x,\lambda)$, where $\alpha = +\infty$ for unreached vertices; min is idempotent and has identity $+\infty$, and `Scatter` again follows the common "signal if and only if `changed`" pattern and is thus regular. Hence, these programs are `TrackFree`, which also explains why Bellman-Ford computes BFS depths correctly on unweighted graphs under streamlined vectorization. In contrast, Integer BFS violates **(I2)**: even with an "inactive" α, `Apply` can still set visited to *True*, changing later branch decisions and signaling, thereby breaking correctness.

Remark. When users believe their algorithm may be `TrackFree`, they provide an algebraic abstraction of `Apply` (capturing the per-round update and the signaling pattern). AutoMI then checks the idempotence conditions; if they hold, it generates track-free streamlined program, otherwise it falls back to the general track-guarded compilation of Section 3.

5. APPLICATIONS

We have already seen both sides of AutoMI via BFS: Integer BFS needs track to be correct (Section 2), while Boolean BFS is `TrackFree` and can drop masks entirely (Section 4). We now highlight additional algorithms that AutoMI converts, and show how the `TrackFree` characterization of Section 4 explains when streamlined vectorization is safe.

ALGORITHM 4: AutoMI generated multi-instance Bellman-Ford(SSSP)

```
1  #define vertex_data { array <int> ans } ;
2  #define msg { array <int> ans } ;

   Function Gather(vertex u, edge (u, v), vertex v)
3      msg.ans ← simd_Add (msg.ans, w(u, v)) ;
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.ans ← simd_Min (msg_acc.ans, msg.ans) ;
7      return msg_acc;
8  Function Apply(vertex v, msg_acc)
9      mask ← simd_CmpGt (v.ans, msg_acc.ans) ;
10     v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans) ;
11     changed ← simd_mask_Set (mask, changed, 1) ;
12 Function Scatter(vertex v)
13     if NOT simd_All_Zero (changed) then // if changed is True for
         some source
14         msg.ans ← simd_Set (msg.ans, v.ans) ;
15         Signal (msg) ;              // activates v's outward neighbors
```

Throughout this section we reuse the notation from Section 4: λ denotes the accumulated (per-vertex) message input to `Apply`, α denotes the corresponding "inactive" message value, and $e_{\mathbf{A}}$ is the `Apply` update function.

SSSP: Bellman-Ford is `TrackFree` (and subsumes BFS on unweighted graphs). Algorithm 4 shows multi-instance Bellman-Ford program generated by AutoMI for SSSP. Each vertex maintains an array v.`ans` storing tentative distances from all sources; initially, v.`ans`$[i]$ is 0 iff $v = s_i$ and $+\infty$ otherwise. In each round, `Gather` adds edge weights to neighbors' distances and `Gather_Acc` takes the minimum; `Apply` then performs the standard relaxation (update if smaller distance is found). `Scatter` signals only when some lane changed.

This program is `TrackFree` because its update rule is idempotent and `Scatter` is regular. In our algebraic abstraction, the accumulated message λ at a vertex is the best candidate distance coming from its neighbors, which is expressible as a matrix-vector product under the tropical semiring:

$$\lambda \;=\; A^T \cdot x \qquad (\oplus = \min,\ \otimes = +),$$

where x corresponds to the current distance vector $\mathbf{x}_{\texttt{ans}}$. The `Apply` update is expressed simply as:

$$e_{\mathsf{SSSP}}(x)\langle\lambda\rangle \;=\; \min(x,\lambda).$$

Now both idempotence conditions from Section 4 hold: (a) its active idempotence since, for fixed λ, $\min(\min(x,\lambda),\lambda) = \min(x,\lambda)$; and (b) its inactive idempotence because the inactive message is $\alpha = +\infty$; hence $\min(x,+\infty) = x$.

In addition, its `Scatter` is regular since a vertex only signals when its state changes in the round (*i.e.,* `changed` is *True*). By Theorem 1, SSSP is `TrackFree`, and AutoMI safely emits the track-free streamlined code in Algorithm 4. On unweighted graphs, Bellman-Ford computes BFS depths, which also explains the remark from Section 2: BFS *can* be safely streamlined, but only for Boolean queries with an algorithm whose update rule satisfies the idempotence test.

Greedy graph coloring. Algorithm 5 shows AutoMI's multi-instance greedy graph coloring [15]. For each source s_i, it computes an independent coloring by iteratively assigning each vertex the minimum color not used by its already-colored neighbors. The algorithm encodes colors compactly

ALGORITHM 5: AutoMI multi-instance Graph Coloring (GC)

```
#define vertex_data { array <bitvec> ans } ;
#define msg { array <bitvec> ans } ;

Function Gather(vertex u, edge (u, v), vertex v)
    return msg ;
Function Gather_Acc(vertex v, msg_acc, msg)
    msg_acc.ans ← simd_Or (msg_acc.ans, msg.ans) ;        // agg neighbors' colors
    return msg_acc;
Function Apply(vertex v, msg_acc)
    mask1 ← simd_CmpNeq (msg_acc.ans, 0) ;    // filter neighbors already colored
    temp ← simd_mask_Add (mask1, msg_acc.ans, 1) ;
    c ← simd_mask_AndNot (mask1, temp, msg_acc.ans) ;
     // min. avail. color
    mask2 ← simd_mask_CmpNeq (mask1, v.ans, c) ;    // sources for new v's color
    v.ans ← simd_mask_Set (mask2, v.ans, c) ;       // update v's color
    changed ← simd_mask_Set (mask2, changed, 1)
Function Scatter(vertex v)
    if NOT simd_All_Zero (changed) then    // if changed is True for some source
        msg.ans ← simd_Set (msg.ans, v.ans) ;      // send out v's color for next rd.
        Signal (msg) ; // activates v's outward neighbors & send msg
```

as bit-vectors: a bit-vector value y indicates which colors appear in the neighborhood, and the minimum available color can be computed by a standard bit trick $(y+1)\& \sim y$.

Despite the more involved `Apply` logic, the same `TrackFree` reasoning applies. Let x denote the per-vertex color-state vector (bit-vectors) and let λ denote the aggregated neighbor color-state, computed via bitwise OR:

$$\lambda \;=\; A^T \cdot x \qquad (\oplus = \text{OR},\ \otimes = \times).$$

`Apply` is abstracted as $e_{\mathsf{GC}}(x)\langle\lambda\rangle = h(x,\lambda)$, where h returns x when no neighbor is colored ($\lambda = 0$), and otherwise returns minimum available color derived from λ (as in Algorithm 5).

Two facts make it `TrackFree`: (i) for fixed λ, the value computed by $h(x,\lambda)$ is stable after one application, so reapplying with the same λ has no further effect (active idempotence); and (ii) the inactive message is $\alpha = 0$, so $h(x,0) = x$ (inactive idempotence). Finally, the program signals only on updates, which implies that `Scatter` is regular. Therefore, graph coloring satisfies Theorem 1, and AutoMI can again drop track and generate streamlined vectorized code.

Advanced graph analytics: SpMV, PPR, and CF. Beyond traversals, AutoMI supports analytics kernels that are naturally written as vertex-centric iterative updates, including Sparse Matrix-Vector multiplication (SpMV), Personalized PageRank (PPR), and Collaborative Filtering (CF).

SpMV and PPR (`TrackFree`). We follow [20, 31, 14, 34, 22] to reduce SpMV to a graph traversal problem; similarly for PPR as an extension of SpMV. Specifically, SpMV computes an output vector $y = Ax$ (one value per vertex) by aggregating neighbor contributions; PPR performs repeated SpMV-style propagation with a source-specific restart and personalization distribution [28]. In GAS implementations of these kernels, `Apply` typically overwrites the current per-vertex value with a function of the accumulated message (and possibly a per-source constant, for restart). As a result, for fixed λ the `Apply` update is stable after one execution (satisfying **(I1)**), and the inactive message corresponds to the aggregation identity (satisfying **(I2)**). Moreover, their `Scatter` follows a regular pattern, *i.e.,* signal-on-change. Hence, SpMV and PPR are `TrackFree` under the criterion, and AutoMI can generate streamlined vectorized multi-instance code.

Table 3: Average speedup of AutoMI, Quegel and MultiLyra over PowerLyra (averaged over `LiveJournal` [5], `Twitter` [8], `Friendster` [1], `UKDomain` [9], `MovieLens` [6], `Netflix` [7], up to 3 billion of edges; ✗ marks algorithms not available.)

system	bBFS	iBFS	SSSP	GC	SpMV	PPR	CF
PowerLyra [13]	1	1	1	1	1	1	1
Quegel [37, 40]	2.32x	2.25x	1.77x	✗	✗	✗	✗
MultiLyra [26]	6.07x	5.53x	5.16x	✗	✗	✗	✗
AutoMI	**14.82x**	**9.51x**	**14.09x**	**6.37x**	**7.79x**	**8.21x**	**3.78x**

Collaborative filtering (not `TrackFree`). In contrast, CF uses Stochastic Gradient Descent [19] to iteratively update latent factors of vertices being visited, starting from a source. Its incremental parameter updates are however not idempotent: repeating `Apply` with the same incoming signal continues to change latent factors, violating **(I1)**. Therefore `TrackFree` is unsafe for CF, and AutoMI falls back to the general track-guarded compilation from Section 3 to preserve correctness.

Takeaway. These examples highlight AutoMI's practical impact: without changing the underlying GAS engine, AutoMI compiles existing single-instance vertex-centric programs into efficient multi-instance SIMD implementations. When misaligned activations would break correctness, AutoMI automatically injects and propagates track to enforce per-source semantics (Section 3); when the idempotence conditions of Section 4 hold, it drops masks entirely and emits streamlined code (`TrackFree`) for even higher performance.

In short, developers can write (or reuse) standard GAS algorithms once, while AutoMI decides how to vectorize them *safely* and delivers multi-source speedups without hand-engineered multi-instance implementations.

6. EXPERIMENTAL STUDY

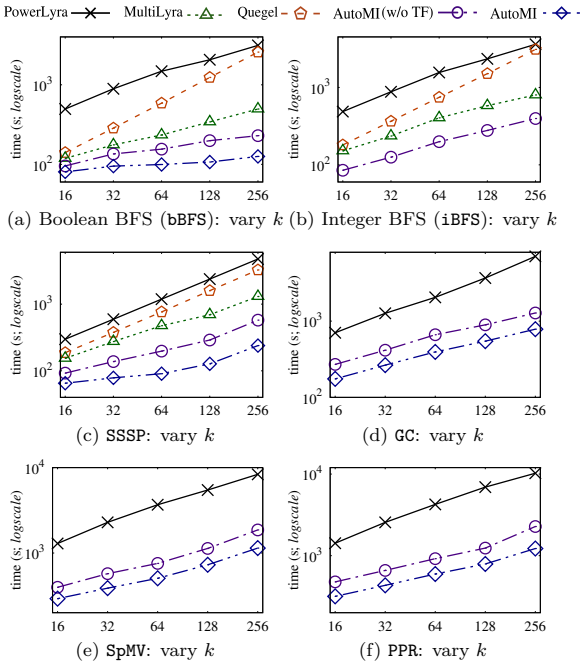
We summarize the key experimental findings of the full paper [41], focusing on (i) end-to-end speedups over state-of-the-art distributed vertex-centric baselines, and (ii) the practical contribution of SIMD and `TrackFree`.

6.1 Implementation and Evaluation Setup

System. AutoMI is a source-to-source compiler for C++ GAS programs. It uses libclang [4] to parse a single-instance program into an AST, applies the annotation rules of Section 3, and then emits a compilable multi-instance C++ program by populating vertex/message fields into length-k arrays and vectorizing the resulting operations. When the `TrackFree` criterion of Section 4 holds, AutoMI generates the streamlined variant that omits track and uses unmasked SIMD. The generated code includes the populated data types and their serialization routines (needed for distributed message passing), and runs on standard GAS engines without runtime changes (PowerLyra [13] by default).

To make SIMD effective, AutoMI stores populated property contiguously with aligned allocation, and maps common operators to Intel SIMD intrinsics [3]; unsupported cases fall back to a small "abstract SIMD" loop (*e.g.,* bit-vector operations in greedy coloring are applied over 256-bit chunks).

Compared systems. We compare against PowerLyra [13] (serially running the single-instance program concurrently over



(a) Boolean BFS (`bBFS`): vary k (b) Integer BFS (`iBFS`): vary k

(c) `SSSP`: vary k (d) `GC`: vary k

(e) `SpMV`: vary k (f) `PPR`: vary k

Figure 3: Running time of multi-instance `bBFS`, `iBFS`, `SSSP`, `GC`, `SpMV`, `PPR` and `CF` with varying number (k) of queries. Results are over `Friendster` [1] graph with 68,349,466 vertices and 2,586,147,869 edges; AutoMI(w/oTF) denotes AutoMI without `TrackFree` optimization, *i.e.,* always using track.

all threads, once per source), MultiLyra [26] (authors' hand-written multi-instance BFS/SSSP atop the same PowerLyra engine), and Quegel [37, 40] (batch query processing; adapted to point-to-all evaluation). Since AutoMI, PowerLyra, and MultiLyra share the same engine, their comparison isolates the effect of code generation, SIMD, and `TrackFree`; Quegel serves as a representative batching baseline.

Workloads and configuration. We use six real graphs, up to 3.3 billion edges (see [41] for details). For `SSSP`, edge weights are sampled uniformly from $[1, \log|V|)$. For each graph, we generate $k \in \{16, 32, 64, 128, 256\}$ sources by sampling a seed vertex and selecting k vertices reached by a BFS from the seed; results are averaged over three seeds. We evaluate `bBFS`, `iBFS`, and `SSSP` on all systems; `GC`, `SpMV`, `PPR`, and `CF` are evaluated with AutoMI and PowerLyra only (not available in MultiLyra/Quegel). All experiments run on four AWS r6i.8xlarge instances (32 vCPUs, 256GB RAM each) connected by a 12.5Gbps network.

6.2 Performance of Vectorized Programs

Overall performance. Table 3 reports the average speedups over PowerLyra across all graphs for each system, and Fig. 3 depicts the detailed evaluation time of all methods over billion-edge graph `Friendster` [1] with varying number k of instances. The results confirm that AutoMI is effective, and is consistently the best among all for each and every case tested. As shown in Table 3, multi-instance GAS programs produced by AutoMI significantly outperform the serial evaluation by PowerLyra, highly optimized MultiLyra code and Quegel batch evaluation for all cases. It is up to 27.83×, 16.42×, 29.51×, 9.84×, 10.26×, 10.65× and 6.17× faster than PowerLyra for `bBFS`, `iBFS`, `SSSP`, `GC`, `SpMV`, `PPR` and `CF`, respectively. Compared to MultiLyra and Quegel, AutoMI is up to 3.82× and 26.35× faster for `bBFS`, 2.57× and 7.12× for `iBFS`, 4.63× and 23.01× for `SSSP`.

Table 4: Run time speedup over PowerLyra, breakdown and `#comms` for `SSSP` over `UKDomain` [9] with 256 queries

system	speedup	`Gather` (s)	`Apply` (s)	`Scatter` (s)	Sync (s)	`#comms`
PowerLyra	1	1787.14	1324.96	2033.49	304.67	4.61×10^{10}
MultiLyra	3.07x	801.48	777.87	183.55	12.25	4.73×10^{8}
AutoMI$_{\text{w/o TF}}$	5.46x	438.89	416.98	139.79	12.24	4.73×10^{8}
AutoMI	11.32x	192.84	196.16	83.68	11.81	4.73×10^{8}

Breakdown. To isolate where AutoMI gains its speedups, Table 4 reports a per-phase breakdown for `SSSP` on `UKDomain` [9] with 256 sources, together with the total number of messages (`#comms`). We also report AutoMI$_{\text{w/o TF}}$, the AutoMI variant that always using the generic track-guarded vectorization.

All multi-instance methods sharply reduce communication: `#comms` drops from 4.61×10^{10} (PowerLyra, serial) to 4.73×10^{8}, and sync time shrinks from 304.67s to about 12s. With the same `#comms` as MultiLyra, AutoMI (w/o TF) is 1.78× faster overall, highlighting the benefit of SIMD. Enabling `TrackFree` removes track overhead and yields a further 2.07× over AutoMI (w/o TF), giving 11.32× speedup over PowerLyra (and 3.69× over MultiLyra).

These reveal three sources of performance gains of AutoMI: (1) *multi-instance interleaving* reduces redundant traversal and communication; (2) *SIMD vectorization* accelerates per-vertex computation relative to handwritten multi-instance code (*e.g.,* MultiLyra); and (3) for `TrackFree` kernels, dropping track removes the mask maintenance overhead.

7. CONCLUSION

Multi-source graph workloads are ubiquitous, yet SIMD-accelerated multi-instance implementations are notoriously hard to write correctly, especially in vertex-centric distributed environment. AutoMI makes this practical: it automatically compiles a standard single-instance GAS program into a correct multi-instance SIMD program by propagating a lightweight per-message mask (track) and using it to guard vectorized updates. Moreover, AutoMI goes beyond "always-mask" compilation: its `TrackFree` optimization and algebraic characterization identify when masks can be dropped entirely, yielding streamlined code with substantially higher performance. Putting together, AutoMI turns multi-instance algorithm development and SIMD efficiency from a manual engineering burden into a compiler-driven capability.

A natural next step is to have broader language and runtime support. While many GAS programs are loop-free and fit the current design, supporting richer control flow, additional vertex-centric interfaces beyond GAS, and alternative execution semantics (*e.g.,* asynchronous) would broaden applicability. Another interesting direction is to target SIMT architectures. This requires rethinking the compilation strategy to generate GPU kernels. A particularly interesting goal is an analogue of `TrackFree` for GPUs: characterizing when a multi-instance program can be safely executed with relaxed per-source scheduling or with cheaper warp masking while still converging to the correct fixed point.

8. REFERENCES

[1] Friendster. http://konect.cc/networks/friendster/, Access: 2024.

[2] Giraph. https://giraph.apache.org/, Access: 2024.

[3] Intel® intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, Access: 2024.

[4] libclang. https://clang.llvm.org/, Access: 2024.

[5] LiveJournal. http://snap.stanford.edu/data/soc-LiveJournal1.html, Access: 2024.

[6] MovieLens. http://grouplens.org/datasets/movielens/, Access: 2024.

[7] Netflix. http://konect.cc/networks/netflix/, Access: 2024.

[8] Twitter. http://konect.cc/networks/twitter/, Access: 2024.

[9] UK domain. http://konect.cc/networks/dimacs10-uk-2007-05/, Access: 2024.

[10] Z. Abul-Basher. Multiple-query optimization of regular path queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1426–1430. IEEE Computer Society, 2017.

[11] K. Ammar and M. T. Özsu. Experimental analysis of distributed graph systems. *Proc. VLDB Endow.*, 11(10):1151–1164, 2018.

[12] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 139–150. IEEE Computer Society, 2003.

[13] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[14] M. K. Esfahani, P. Kilpatrick, and H. Vandierendonck. Exploiting in-hub temporal locality in spmv-based graph processing. In X. Sun, S. Shende, L. V. Kalé, and Y. Chen, editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 42:1–42:10. ACM, 2021.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[16] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, 2014.

[17] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[18] M. Kaufmann, M. Then, A. Kemper, and T. Neumann. Parallel array-based single- and multi-source breadth first searches on large dense graphs. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 1–12. OpenProceedings.org, 2017.

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[20] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association, 2012.

[21] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 666–677. IEEE Computer Society, 2012.

[22] J. Li, W. Zhao, N. Ntarmos, Y. Cao, and P. Buneman. MITra: A framework for multi-instance graph traversal. *Proceedings of the VLDB Endowment*, 16(10), 2023.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8), 2012.

[24] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: an experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, 2014.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[26] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 349–358. IEEE, 2019.

[27] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2), 2015.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[29] P. Peng, Q. Ge, L. Zou, M. T. Özsu, Z. Xu, and D. Zhao. Optimizing multi-query evaluation in federated RDF systems. *IEEE Trans. Knowl. Data Eng.*, 33(4):1692–1707, 2021.

[30] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proc. VLDB Endow.*,

10(3):121–132, 2016.

[31] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[32] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[33] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.

[34] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos. Accelerating graph analytics by utilising the memory locality of graph partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 181–190. IEEE, 2017.

[35] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.

[36] D. Yan, Y. Bu, Y. Tian, A. Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[37] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.*, 9(7):564–575, 2016.

[38] H. Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10, 2010.

[39] Q. Zhang, H. Chen, D. Yan, J. Cheng, B. T. Loo, and P. Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 40–51, 2017.

[40] Q. Zhang, D. Yan, and J. Cheng. Quegel: A general-purpose system for querying big graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2189–2192. ACM, 2016.

[41] W. Zhao, Y. Cao, P. Buneman, J. Li, and N. Ntarmos. Automating vectorized distributed graph computation. *Proc. ACM Manag. Data*, 2(6):254:1–254:27, 2024.

[42] W. Zhao, J. Li, Y. Cao, and N. Ntarmos. Mitra: Populating graph traversal algorithms. In *ICDE*, pages 4576–4579. IEEE, 2025.