Compiling Graph Applications for GPUs with GraphIt

Ajay Brahmakshatriya
MIT CSAIL
ajaybr@mit.edu

Yunming Zhang
MIT CSAIL
yunming@mit.edu

Changwan Hong
MIT CSAIL
changwan@mit.edu

Shoaib Kamil
Adobe Research
kamil@adobe.com

Julian Shun
MIT CSAIL
jshun@mit.edu

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

***Abstract*—The performance of graph programs depends highly on the algorithm, the size and structure of the input graphs, as well as the features of the underlying hardware. No single set of optimizations or one hardware platform works well across all settings. To achieve high performance, the programmer must carefully select which set of optimizations and hardware platforms to use. The GraphIt programming language makes it easy for the programmer to write the algorithm once and optimize it for different inputs using a scheduling language. However, GraphIt currently has no support for generating high-performance code for GPUs. Programmers must resort to re-implementing the entire algorithm from scratch in a low-level language like CUDA with an entirely different set of abstractions and optimizations in order to achieve high-performance on GPUs.**

We propose G2, an extension to the GraphIt compiler framework, that achieves high performance on both CPUs and GPUs using the same algorithm specification. G2 significantly expands the optimization space of GPU graph processing frameworks with a novel GPU scheduling language and compiler that enables combining load balancing, edge traversal direction, active vertexset creation, active vertexset processing ordering, and kernel fusion optimizations. G2 also introduces two performance optimizations, Edge-based Thread Warps CTAs load balancing (ETWC) and EdgeBlocking, to expand the optimization space for GPUs. ETWC improves load balancing by dynamically partitioning the edges of each vertex into blocks that are assigned to threads, warps, and CTAs for execution. EdgeBlocking improves the locality of the program by reordering the edges and restricting random memory accesses to fit within the L2 cache. We evaluate G2 on 5 algorithms and 9 input graphs on both Pascal and Volta generation NVIDIA GPUs, and show that it achieves up to 5.11× speedup over state-of-the-art GPU graph processing frameworks, and is the fastest on 66 out of the 90 experiments.

***Index Terms*—Compiler Optimizations, Graph Processing, GPU, Domain-Specific Languages**

I. Introduction

Graph processing is at the heart of many modern applications, such as recommendation engines [1], [2], social network analytics [3], [4], and map services [5]. Achieving high performance is important because these applications often need to process very large graphs and/or have strict latency requirements [1].

In prior work, we built the GraphIt DSL compiler [6], [7] to generate high-performance CPU code from a high-level algorithm language. GraphIt achieves state-of-the-art performance on CPUs across different algorithms and graph inputs by introducing a scheduling language to tune optimizations. The algorithm language has primitives for topology-driven algorithms, data-driven algorithms, and priority-based algorithms. This algorithm and schedule representation makes it easy for the programmer to write an algorithm once and generate different highly-optimized implementations by simply changing the schedule. We will discuss in detail the GraphIt algorithm and scheduling language with examples in Section IV.

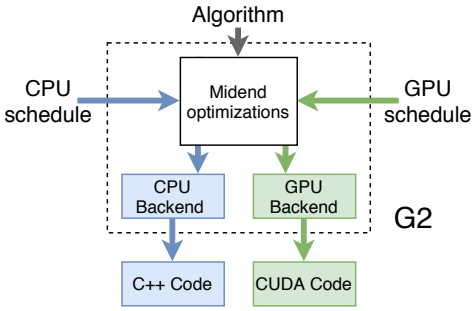

Fig. 1: G2 adds a new GPU backend to GraphIt that produces CUDA code from the same GraphIt algorithm language and a scheduling language tailored for GPU optimizations.

Apart from a large body of work for optimizing algorithms on different inputs on CPUs [6], [8], [9], [10], [11], [12], [13], [14], [15], researchers have also used different hardware platforms for high-performance graph processing, including GPUs [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], and we have found that no single hardware is suitable for all algorithms. Some algorithms perform better on CPUs and others perform better on GPUs. Shared-memory CPUs have out-of-order execution, larger caches, and larger memories than GPUs, which enables larger graphs to be processed. At the same time, GPUs typically have an order of magnitude more computing power and memory bandwidth [26], and can better exploit data parallelism.

Unfortunately, since GraphIt does not support code generation for GPUs, programmers have to resort to writing CUDA implementations from scratch while building completely new

abstractions and optimizations which rarely can be applied to different algorithms. Therefore, it is critical to build a common framework that can use a single algorithm representation to generate both CPU and GPU code, and that also supports compiler transformations and compiler analyses specific to their respective platforms.

We introduce G2, an extension to the GraphIt DSL compiler to generate high-performance implementations on both CPUs and GPUs from the same high-level graph algorithm specification. GraphIt already demonstrated that separating algorithm specification from scheduling representation is a powerful way to optimize graph algorithms. We apply the same idea to GPUs by using a scheduling language to choose optimizations tailored specifically for GPUs. Figure 1 shows the new architecture of the GraphIt compiler with the G2 extensions. Generating high-performance GPU implementations presents significantly more challenges. GPUs incur a huge cost when accessing global memory in an uncoalesced fashion across threads in a warp. We have to optimize not just for temporal locality, but also for locality across threads in a warp. Achieving load balancing is much harder on GPUs than on CPUs due to a large number of execution units. The GPU implementations must also handle data transfer between the host and the device depending on what portion of the code executes on the GPU.

Numerous performance optimizations have been proposed to address the challenges of using GPUs for graph processing [16], [27], [28]. However, no existing GPU graph processing framework supports all of the necessary optimizations to achieve high performance, as shown in Table I. GPU graph processing libraries [24], [19], [20] cannot easily support optimizations that require global program and data structure transformations. A compiler approach that employs program transformations and code generation is more flexible. However, existing compiler approaches [17] lack analysis passes, code generation, and runtime library support for optimizations such as direction-optimization and active vertexset creation.

G2 proposes a novel scheduling language, new program analyses, and a code generation algorithm to support the wide variety of performance optimizations while maintaining a simple high-level algorithm specification. The GPU scheduling language enables programmers to apply load balancing, edge traversal direction, active vertexset creation, active vertexset processing ordering, and kernel fusion optimizations. The compiler uses a liveness analysis to reuse allocations for some variables to improve memory efficiency. G2 performs program analyses and transformations on user-defined functions (UDFs) and iterative edge traversals to incorporate optimizations that require global program transformations, such as direction-optimization and kernel fusion. Figure 4 the GPU scheduling language input for the same BFS algorithm shown in Figure 2.

We also present two performance optimizations, Edge-Blocking and Edge-based Thread Warps CTAs (ETWC), to further expand the space of graph optimizations on GPUs. EdgeBlocking reorders the edges in coordinate format (COO) to group together edges accessing segments of vertices that fit in the L2 cache of the GPU. This optimization improves performance by keeping random memory accesses within the fast L2 cache, reducing slower global memory accesses. ETWC partitions the edges of active vertices into small, medium, and large buckets that can be processed by groups of threads, warps, and Cooperative Thread Arrays (CTAs), respectively, to improve load balancing and utilization of Streaming Multiprocessors (SMs).

We evaluate the performance of G2 on five graph algorithms: PageRank (PR), Breadth-First Search (BFS), Delta-Stepping for Single-Source Shortest Paths (SSSP), Connected Components (CC), and Betweenness Centrality (BC). G2 is built as an extension to GraphIt [6], [7]. We are able to obtain up to 5.11× speedup over the fastest of the state-of-the-art GPU graph frameworks Gunrock [24], GSwitch [19], and SEP-graph [20]. G2 generates CPU implementations that match the performance of the original GraphIt compiler, thus not compromising CPU performance.

This paper makes the following contributions:

- An analysis of the fundamental tradeoffs among locality, work-efficiency, and parallelism in GPU graph optimizations (Section III).
- A novel GPU scheduling language that allows programmers to combine load balancing, edge traversal direction, active vertexset creation, active vertexset processing ordering, and kernel fusion optimizations (Section V).
- A compiler with program analyses and transformations to support efficient code generation on both CPU and GPU platforms (Sections IV and VI).
- Two performance optimizations for GPUs— EdgeBlocking, which improves the locality of edge processing, and ETWC, which improves load balancing (Section III).
- A comprehensive evaluation of G2 on GPUs showing that G2 outperforms state-of-the-art GPU graph processing frameworks by up to 5.11× on 90 experiments across two generations of NVIDIA GPUs (Section VII).

II. Related Work

Compilers for Graph Programs. G2 is built as an extension to our previous work GraphIt [6], [7], which is a domain-specific language and compiler that expands the optimization space to outperform other CPU frameworks by decoupling algorithms and optimizations. Apart from adding support for GPUs, G2 also adds new mid-end optimization passes, which can potentially be reused for other architectures. G2 creates a novel GPU scheduling language that enables users to explore a much wider space of optimizations for GPUs.

IrGL [17] is another compiler framework that introduces a programming language and an intermediate representation specifically for graph applications on GPUs. IrGL first proposed the kernel fusion optimization, which reduces the overhead of kernel launch for SSSP and BFS on high-diameter road networks. However, IrGL supports a much narrower space of optimizations (see Table I). Unlike G2, IrGL does not support code generation for CPUs and GPUs from the same high-level representation.

Framework	**G2**	GSWITCH	SEP-Graph	Gunrock	IrGL
Load Balancing	WM, CM, ETWC, TWC, Strict, VertexBased, EdgeOnly	CM, WM, Strict, TWC	CM	VertexBased, TWC, EdgeBased	Serial, TB, Warp, FineGrained, TB+Warp, TB+FG, WP+FG, TB+Warp+FG
Edge Blocking	Supported	Not-Supported	Not-Supported	Not-Supported	Not-Supported
Vertexset Creation	SparseQueue, Bitmap, Boolmap	SparseQueue, Bitmap	SparseQueue	SparseQueue, Bitmap	SparseQueue
Kernel Fusion	Supported	Not-Supported	Supported	Not-Supported	Supported
Direction-Optimization	Supported	Supported	Supported	Supported	Not-Supported
Vertexset Deduplication	Supported	Not-Supported	Supported	Supported	Not-Supported
Vertices Ordering	Supported	Supported	Supported	Supported	Supported
Total Opt. Combinations	**576**	32	16	48	32

TABLE I: Number of available optimizations in state-of-the-art GPU graph processing frameworks. Total is the product of all of the options.

Graph Frameworks for both CPUs and GPUs. GraphBLAS [29], [30] uses sparse linear algebra operations as building blocks for graph algorithms. The GraphBLAS approach does not support ordered algorithms, such as Delta-Stepping for SSSP [31], which are needed for achieving high performance on SSSP and other problems [19], [24], [20]. G2 supports ordered graph algorithms with a GPU-based two-bucket priority queue. Abelian [32] uses the Galois framework as an interface for shared-memory CPU, distributed-memory CPU, and GPU platforms. However, it lacks support for direction-optimization, various load balancing optimizations, and active vertexset creation optimizations, which are needed to achieve high performance.

Graph Frameworks for GPUs. There has been a tremendous amount of effort on developing high-performance graph processing frameworks on GPUs (e.g., [16], [17], [18], [20], [21], [27], [22], [23], [24], [33], [34], [35], [36], [25], [37], [38], [39], [40], [41], [42], [43]). Gunrock [24] proposes a novel data-centric abstraction and incorporates existing GPU optimization strategies. GSWITCH [19] identifies and implements a set of useful optimizations and uses auto-tuning to achieve high performance. DiGraph [44] is another framework that targets multiple GPUs and achieves higher throughput by employing a novel path scheduling strategy that minimizes communication and makes the algorithms converge in fewer iterations. However, most of the frameworks support only a subset of existing optimizations and cannot achieve high performance on all algorithms, graphs, and GPU architectures. (Table I).

Other Graph Processing Frameworks. There has been a large body of work on graph processing for shared-memory multicores (e.g., [8], [9], [10], [45], [11], [12], [13], [46], [47], [32], [48], [49], [14]), distributed memory (e.g., [50], [51], [52], [53], [54], [55], [56], [57], [58]), and external memory (e.g., [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69]). These frameworks support a limited set of optimizations and cannot achieve consistent high performance across different algorithms and graphs [6].

Load Balancing Optimizations on GPU. TWC is a dynamic load balancing strategy and prefix sum-based frontier computation designed for efficient breadth-first search on GPUs [16]. Gunrock [24] and GSwitch [19] implement both TWC and a few other static load balancing techniques, which we describe in more detail in Section III. Tigr [27] and Subway [28] achieve load balancing by preprocessing the graph to ensure that all the vertices have a similar number of neighbors. This approach incurs significant prepossessing overhead and does not generalize across all graph algorithms. Tigr also exposes a much smaller optimization space and does not show how the load balancing techniques can be combined with other optimizations. G2 introduces the ETWC scheme, which reduces the runtime overhead in exchange for some load balance.

Locality-Enhancing Optimizations. The irregular memory access pattern in graph algorithms makes it hard to take advantage of the memory hierarchy on CPUs and GPUs [70], [14]. On CPUs, locality can be enhanced by tiling the graph [14], [45], [51], [8] or by reordering the memory accesses at runtime [71], [72]. Reordering the memory accesses at runtime on GPUs is not practical because there is not enough memory to buffer all of the irregular memory updates and the performance improvement cannot compensate for the extra work to be done. A naive graph tiling approach on GPUs results in poor performance due to insufficient parallelism. EdgeBlocking finds a good balance between locality and parallelism by tiling for the shared L2 cache and processing one tiled subgraph at a time. Previous work [73], [74] has used tiling in sparse matrix computations.

III. GPU Optimization Space

Optimizations that make tradeoffs among locality, parallelism, and work-efficiency (number of instructions) can improve the performance of graphs algorithms by orders of magnitude over naive implementations [16], [24], [19] on GPUs. In this section, we describe the tradeoff space of existing graph optimizations for GPUs.

Active Vertexset Creation. Different ways of creating output frontiers have different tradeoffs. These creation mechanisms include fused [19] (with the edge traversal) vs. unfused and different representations such as bitmaps, bytemaps, and sparse arrays. The fused mode trades off parallelism for better work-efficiency. Bitmaps have better locality but suffer from atomic synchronization overhead, unlike bytemaps.

Kernel Fusion across Iterations. The iterative nature of many graph algorithms can be expressed with a while loop. A GPU kernel can be launched in each iteration in the while loop, but if there is not enough work per iteration, the kernel launch overhead can dominate the running time. To alleviate this overhead, the while loop can be moved into the kernel so that the GPU kernel is launched only once [17]. This improves work-efficiency but sacrifices parallelism and load balancing.

Direction-Optimization. The push (each vertex updates its neighbors) and pull (each destination reads its neighbors

and updates itself) modes for updating vertex data offer a tradeoff between work-efficiency, atomic synchronization overheads, and parallelism. Direction-optimization achieves the best of both worlds by dynamically switching between the two directions based on the frontier size [75], [9], [76].

Active Vertexset Deduplication. Since active vertices may share common neighbors, some vertices can appear multiple times in the next frontier, which may cause redundant computation and even lead to incorrect results. This can be avoided by an explicit deduplication step, which affects work-efficiency, increasing or decreasing it depending on the number of duplicate vertices.

Active Vertexset Processing Ordering. The active vertices can be processed according to a priority-based ordering, which can significantly improve the work-efficiency for ordered graph algorithms such as Delta-Stepping for SSSP, but at the expense of reducing parallelism [31], [7], [19].

Load Balancing. Different load balancing schemes have tradeoffs between parallelism and work-efficiency to various degrees. Warp Mapping (WM) and CTA Mapping (CM) [19], [23] divide the active vertices evenly across different warps and CTAs, and achieve high parallelism but sacrifice work-efficiency due to overheads in partitioning the active vertices. STRICT incurs even higher overheads but ensures that each thread in the GPU processes the same number of edges. TWC splits active vertices into buckets processed by threads, warps, or CTAs based on their degrees [16], [24]. TWC has smaller runtime overhead (high work-efficiency), but a lower degree of parallelism compared to WM, CM, and STRICT. However, for some graphs, this overhead can still be large compared to the speedup achieved with better load balancing. Vertex-Parallel (VP) simply maps each vertex to a thread and has the least overhead but poor load balancing.

In this paper, we introduce two optimizations: ETWC, a load balancing strategy that has reduced runtime overhead as compared to TWC; and EdgeBlocking, a graph partitioning optimization for improving locality of vertex data accesses. Details of these optimizations are described in Section VI.

IV. Algorithm Language

G2 separates the algorithm logic from performance optimizations with a separate algorithm language and scheduling language. Since the goal of G2 is to generate high-performance GPU code with minimum effort while moving from CPUs, we use the same algorithm language from GraphIt which supports unordered graph algorithms [6], along with the extensions in PriorityGraph [7] to support ordered graph algorithms. The algorithm language works with vectors, vertex and edge sets, functional operators over sets, and priority queue operators, and hides all of the low-level details needed for correctness and performance, such as atomic synchronization and bit manipulation, from the programmer.

Figure 2 shows an example of the breadth-first search (BFS) algorithm written in the GraphIt algorithm language. The program starts with a vertexset with just the `start_vertex` and repeatedly updates the neighbors of the vertices in the frontier by applying the `updateEdge` and `toFilter` user-defined functions, while also creating the next frontier from the vertices just updated. This is done till the frontier is empty.

```
element Vertex end
element Edge end
const edges : edgeset{Edge}(Vertex,Vertex)
   = load(argv[1]);
const vertices : vertexset{Vertex}
   = edges.getVertices();
const parent : vector{Vertex}(int) = -1;

func toFilter(v : Vertex) -> output : bool
  output = (parent[v] == -1);
end

func updateEdge(src : Vertex, dst : Vertex)
  parent[dst] = src;
end

func main()
  var frontier : vertexset{Vertex}
     = new vertexset{Vertex}(0);
  var start_vertex : int = atoi(argv[2]);
  frontier.addVertex(start_vertex);
  parent[start_vertex] = start_vertex;
  #s0# while (frontier.getVertexSetSize() != 0)
    #s1# var output : vertexset{Vertex} =
      edges.from(frontier).to(toFilter).
        applyModified(updateEdge, parent, true);
    delete frontier;
    frontier = output;
  end
  delete frontier;
end
```

Fig. 2: BFS program written in the GraphIt algorithm language.

```
configApplyDirection("s0:s1", "DensePull-SparsePush")
configApplyDenseVertexSet("s0:s1", "src-vertexset",
   "bitvector", "DensePull")
configApplyParallelization("s0:s1",
   "dynamic-vertex-parallel")
```

Fig. 3: GraphIt scheduling language input for optimizing the BFS algorithm in Figure 2 on CPUs using a hybrid traversal

```
SimpleGPUSchedule s1;
s1.configDirection(PUSH);
s1.configLoadBalance(VERTEX_BASED);

SimpleGPUSchedule s2 = s1;
s2.configDirection(PULL, BITMAP);
s2.configDeduplication(DISABLED);
s2.configLoadBalance(VERTEX_BASED);
s2.configFrontierCreation(UNFUSED_BITMAP);

HybridGPUSchedule h1(VERTEXSET_SIZE, "argv[3]", s1, s2);
program->applyGPUSchedule("s0:s1", h1);
```

Fig. 4: G2 scheduling language input for optimizing the BFS algorithm in Figure 2 on GPUs using a hybrid traversal. Since GPUs requires completely different transformations, the optimizations enabled are different from those in Figure 3.

Figure 3 shows the scheduling language input used in GraphIt for optimizing this algorithm on CPUs with direction-optimization (hybrid traversal). The scheduling input enables hybrid traversal with bitvector representation for the source vertex frontier when traversing in the PULL direction. This schedule also enables dynamic vertex-based parallelism. We have modified GraphIt in such a way that preserves the scheduling input for CPUs without significant change in the generated code. Figure 4 shows the scheduling input used in G2 for the BFS algorithm when generating code for GPUs. This schedule also enables hybrid traversal but also tunes various

GPU specific scheduling options such as deduplication, load balancing, and active vertexset creation strategy. These options will be explained in detail in Section V.

V. Scheduling Language

We propose a novel scheduling language for GPUs that allows users to combine the load balancing, traversal direction, active vertexset management, and work-efficiency optimizations described in Section III. These optimizations are different from the ones for CPUs because GPUs have drastically different hardware features, such as a larger number of threads, larger memory bandwidth, smaller L1 and L2 cache per thread, smaller total memory, and different synchronization overheads.

The scheduling language supports two main types of schedules, `SimpleGPUSchedule` and `HybridGPUSchedule`. The programmer can use them together to create complex schedules that are best suited for their algorithm and graph type.

The `SimpleGPUSchedule` object directly controls the scheduling choices related to load balancing, traversal direction, active vertexset management, and work-efficiency. As shown in Table II, the objects have six `config*` functions. The programmer can use these functions to specify the load balancing strategy, the edge traversal direction, the output frontier creation strategy, whether deduplication is enabled, the delta value for priority queues, and whether kernel fusion is applied to a particular loop along with some optional parameters.

The `HybridGPUSchedule` objects combine two `SimpleGPUSchedule` objects with some runtime condition. The two `SimpleGPUSchedule` objects can be entirely different, using different load balancing schemes, frontier creation types, traversal directions, etc. Depending on whether the runtime condition evaluates to true or false, one of the two `SimpleGPUSchedule` objects is invoked. This API enables the programmer to implement complex optimizations like direction-optimization by combining two `SimpleGPUSchedule` objects (Table III).

The scheduling language shares some key features with the existing CPU scheduling language, such as the ability to choose the edge traversal direction, whether deduplication is enabled, and whether the iteration is vertex-parallel or edge-parallel (based on load balancing type). These features are utilized by the high-level compiler to perform target-independent optimizations.

Scheduling for BFS. Figure 5 shows three different schedules that can be applied to the same BFS program. The first example shows the default `PUSH` schedule highlighted in green. Above the schedule, we show snippets of the corresponding generated CUDA code from G2. Notice that even with the default schedule, the UDF (`updateEdge`, highlighted in blue) has been transformed to use an atomic compare-and-swap instruction when updating the destination vertex data. This is because atomics are required here for correctness, not for optimization.

The second example schedule shows how kernel fusion can be applied to an entire loop body. Notice that in the generated code, the entire while loop is moved to the CUDA kernel and the main function just calls this kernel. This schedule is suitable for high-diameter graphs that have low parallelism within a round (e.g., road networks).

The third example shows a direction-optimizing schedule. We create two individual `SimpleGPUSchedule` objects, `s1` and `s2`, and configure one for the `PUSH` direction and the other for the `PULL` direction, along with other parameters. We then combine the two into a single `HybridGPUSchedule` object that chooses between the two based on the size of the input frontier and apply this to the `applyModified` operator. The generated code has a runtime condition based on the size of the vertexset that chooses between two separate implementations. Two versions of the UDF are created because the edge traversal does not require atomics when iterating in the `PULL` direction. This schedule is suitable for power-law graphs (e.g., social network graphs) that have significantly different numbers of active vertices in different rounds.

VI. Compiler Implementation

This section details the changes that G2 makes to the GraphIt compiler to support the new GPU backend. This includes analyses and transformations added to the mid-end and the backend for generating code for both the host and device using the parameters from the new GPU scheduling language. We also describe the implementations of the EdgeBlocking and ETWC optimizations.

A. Liveness Analysis for Frontier Reuse

G2 adds a new liveness analysis to the mid-end to reduce costly memory allocations on GPUs. The `edges.applyModified` function generally needs to allocate memory for the frontier that it outputs, but if the frontier it takes as an input is not required anymore, the allocated memory can be reused to avoid calls to the costly `cudaMalloc` and `cudaFree` functions (particularly useful when called inside a loop). To determine whether the memory for frontiers can be reused, the compiler performs a liveness analysis on the variables in the `main` function. If the analysis shows that the input frontier is deleted before being used again, it sets the `can_reuse_frontier` flag in `applyModified`.

B. UDF Transformations

G2 implements a dependence analysis pass to automatically insert atomic instructions for updates to vertex data in UDFs. Whether an update requires atomics depends on which set of vertices are iterated over in parallel by different threads, which in turn depends on the load balancing strategy used. Therefore, the analysis pass needs to be made aware of the different types of parallelism offered by the load balancing strategies added by the GPU scheduling language. The analysis combines the results of the dependence analysis and read/write analysis to identify updates that are shared by multiple threads and inserts atomics.

Apart from inserting atomics, the compiler also transforms the UDF to enqueue the vertices into the output frontier when

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

TABLE II: Description of the SimpleGPUScheduling type and associated `config` functions. The default value for each parameter is in **bold**.

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

TABLE III: Description of the HybridGPUScheduling type and constructor. Default value for each parameter is shown in **bold**.

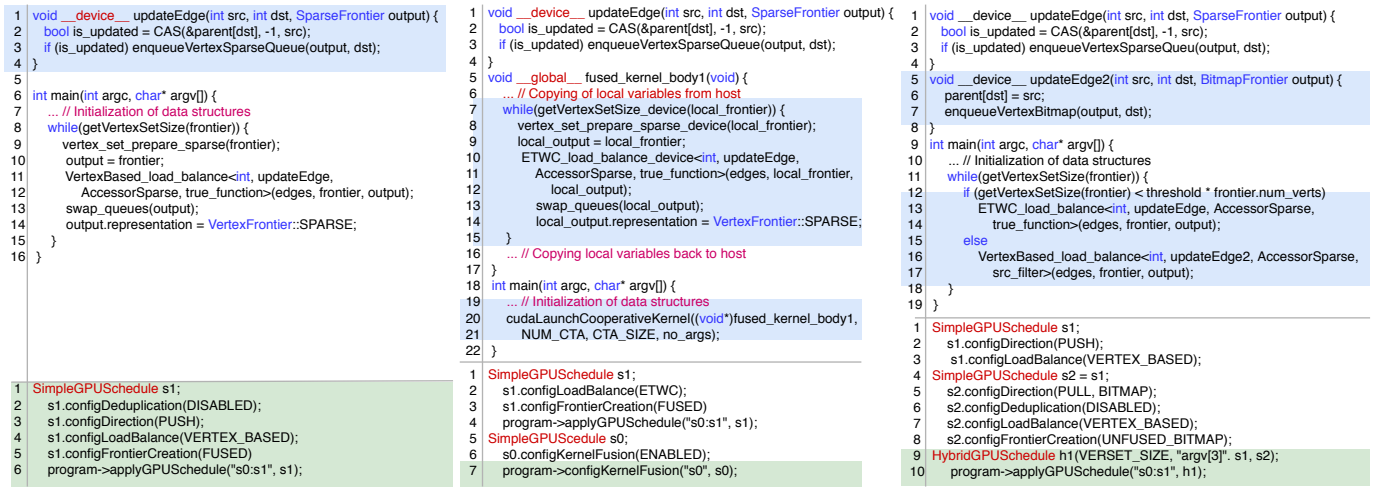

Fig. 5: Different schedules for a default `PUSH` based implementation (left), an implementation with kernel fusion enabled (middle), and an implementation with direction-optimization (right). The code at the top shows snippets of the generated CUDA code for each of the schedules.

their vertex data is updated. The compiler refers to the schedule and the tracking variable to select the enqueue function to use.

C. GPU Backend Implementation

The GPU backend performs transformations and optimizations required for generating CUDA code, including transfer of data between the host and device, load balancing for a hierarchy of threads, inter-thread and inter-warp communication, and kernel fusion. The GPU backend generates host code that takes care of loading graphs, allocating data structures, and executing outer loops. It also generates device code that actually implements the edge/vertexset iteration operators.

The GPU backend also implements the kernel fusion optimization. As shown on Line 23 of Figure 2, we can annotate an entire loop to be fused using a label. The programmer can then enable kernel fusion with the `configKernelFunction` function. With this schedule, the compiler must launch a single CUDA kernel for the entire loop. This means that all steps inside the body of the loop must execute on the device. G2 first performs an analysis to figure out if the body contains a statement that it cannot execute on the device (e.g., creation of objects that require allocation of memory or a call to the `delete` operator, if the hardware/runtime does not support dynamic memory allocation).

G2 generates code for the fused kernel by iterating over the program AST multiple times. In the first pass, the backend generates a `__global__` kernel that has the actual while loop and calls to the functions for the operators inside the loop. Here, the compiler also handles local variables referenced from the `main` function by hoisting them to `__device__` global variables. Usually, the operator requires a lot more threads than the ones launched for the single kernel. G2 inserts another `for` loop around the operator code so that each thread can do

work for more threads serially. In the second pass, G2 actually generates a call to the CUDA kernel where the `while` loop is in the `main` function. It also copies all required local variables to the newly-hoisted global variables before calling the kernel and then copies them back after the kernel returns.

Load Balancing Library. Effective load balancing is one of the key factors for obtaining high performance on GPUs. By ensuring that each thread has an almost equal number of edges to process, we minimize the time that threads are waiting for other threads to finish. Load balancing within a warp is even more critical to minimize divergence in some generations of GPUs where diverged threads are executed serially. However, precise load balancing comes at a cost. Before actually processing the edges, threads have to coordinate to divide work among CTAs, warps, and threads, which involves synchronization, global memory accesses, and extra computations. Different load-balancing schemes provide a tradeoff between cost and the amount of balancing achieved. Which load balancing strategy is the most beneficial depends on the algorithm and the graph input. As a result, G2 currently adds support for a total of 7 load balancing strategies. A load-balancing library creates an abstraction with a templated interface that processes a set of edges after dividing them between threads. This modular design not only makes code generation easy but also makes it easy to add more load balancing techniques in the future. The core library has both `__global__` and `__device__` wrappers allowing these routines to be reused inside the fused kernel when kernel fusion is enabled.

D. GPU-Specific Optimizations

This section explains the two optimizations (EdgeBlocking and ETWC) G2 adds to the GraphIt compiler to boost performance on GPUs by improving cache utilization and load balancing, respectively.

EdgeBlocking. We propose the new EdgeBlocking optimization, which tiles the edges into a series of subgraphs to improve the locality of memory accesses. Algorithm 1 shows the steps for preprocessing an input graph to apply the EdgeBlocking optimization. The preprocessing is a two-step process. The first for-loop (Line 6–8 of Algorithm 1) iterates through all of the edges and counts the number of edges in each subgraph. The algorithm then uses a prefix sum on these counts to identify the starting point for each subgraph in the output graph's edges array. The second for-loop (Line 10–14 of Algorithm 1) then iterates over each edge again and writes it to the appropriate subgraph while incrementing that subgraph's counter. We apply the function `process_edge` to each edge as shown in Algorithm 2. The arguments to this function are the source vertex and the destination vertex. The pseudocode shown in Algorithm 2 is executed by each thread in a thread block. We use an outer for-loop (Line 2 of Algorithm 2) that iterates over each subgraph. Within each subgraph, the edges are then processed by all of the threads using a `cooperative for` (Line 5 of Algorithm 2). All of the threads are synchronized between iterations over separate subgraphs to avoid cache interference. EdgeBlocking improves the performance of some algorithms by up to 2.94×, as we show in Table X.

Algorithm 1: Algorithm for preprocessing the input graph when applying the EdgeBlocking optimization. The algorithms takes as input a graph in the Coordinate (COO) format and the blocking size. The output is a new graph in COO format with the edges blocked.

```
1: Input: Number of vertices per segment N, Graph G in COO format.
2: Output: Graph Gout in COO format.
3: numberOfSegments ← ceil(G.num_vertices/N)
4: segmentSize[numberOfSegments + 1] ← 0
5: Gout ← new Graph
6: for e : G.edges do
7:     segment ← floor(e.dst/N)
8:     segmentSize[segment] ← segmentSize[segment] + 1        ▷Count the number of edges in each subgraph.
9: prefixSum ← prefix_sum(segmentSize)        ▷Identify the starting point for each subgraph using prefix sum.
10: for e : G.edges do
11:     segment ← floor(e.dst/N)
12:     index ← prefixSum[segment]
13:     Gout.edges[index] ← e        ▷Add edge to the appropriate subgraph.
14:     prefixSum[segment] ← prefixSum[segment] + 1
15: Gout.numberOfSegments ← numberOfSegments
16: Gout.segmentStart ← prefixSum
17: Gout.N ← N
```

Algorithm 2: Implementation of the *edgeset.apply* operator with the EdgeBlocking optimization. Here, the input graph is preprocessed and the edges are blocked. The function *process_edge* is applied to each edge in the graph. Note that the EdgeBlocking optimization can be applied only when all the edges in the graph are being processed.

```
1: Input: Graph G in COO format,
2: for segIdx : G.numberOfSegments do        ▷Iterate over the subgraphs.
3:     start_vertex ← G.N * segIdx
4:     end_vertex ← G.N * (segIdx + 1)
5:     coop for eid in G.segmentStart[segIdx − 1] : G.segmentStart[segIdx] do
6:         e ← G.edges[eid]
7:         process_edge(e.src, e.dst)
8:     sync_threads()
```

Edge-based Thread Warps CTAs (ETWC). Inspired by load balancing schemes in previous work [25], [17], [37], [35], [23], [16], ETWC is a load balancing strategy that further reduces the runtime overhead of TWC by performing TWC style assignment within each CTA instead of across all CTAs. Similar to CM, each CTA starts with an equal number of vertices. Within each CTA, ETWC partitions edges of the vertices into chunks that are processed by the entire CTA, a warp, or an individual thread. These partitions are processed in separate stages to minimize divergence.

Algorithm 3 shows the ETWC load balancing strategy. Each thread has a unique ID (idx), and 3 queues ($Q[0]$, $Q[1]$, $Q[2]$). These 3 queues correspond to the edges to be processed in the 3 stages. The outgoing edges of each vertex are partitioned and added to these queues such that each partition is an exact multiple of the chunk size starting from the greatest chunk size (CTA size) (Line 10–21). This ensures that fewer edges are processed in the individual thread stage, which is prone to divergence.

For each stage, the representative thread (the first thread in the CTA or warp, or the thread itself) dequeues the 3-tuple from the corresponding queue and broadcasts it to the other threads in the CTA or warp (Line 24–26) for cooperatively

Algorithm 3: Pseudocode for the implementation of ETWC load balancing strategy.

```
1: Input: Graph G in CSR format, input frontier (input_frontier), and three queues
   Q[0, 1, 2], and the number of threads to process edges of a vertex in three stages
   Stage_gran[0, 1, 2].
2: idx = threadblockID * threadblockSIZE + threadID
3: Initialize Q[0, 1, 2]
4: if idx < input_frontier.size then
5:     src_id ← getFrontierElement(input_frontier, idx);
6:     size ← G.rowptr[src_id + 1] − G.rowptr[src_id]    ▷The degree of a
   vertex.
7:     start_pos ← G.rowptr[src_id]
8:     end_pos ← G.rowptr[src_id + 1]    ▷State_gran[2] is usually CTASize.
9:     Stage_elt[2] ← ⌊size/Stage_gran[2]⌋ × Stage_gran[2]
10:    if Stage_elt[2] > 0 then
11:        Q[2].Enqueue({start_pos, start_pos + Stage_elt[2], src_id});
12:        start_pos ← start_pos + Stage_elt[2]
13:        size ← size − Stage_elt[2]
14:    Stage_elt[1] ← ⌊size/Stage_gran[1]⌋ × Stage_gran[1]    ▷WarpSize.
15:    if Stage_elt[1] > 0 then
16:        Q[1].Enqueue({start_pos, start_pos + Stage_elt[1], src_id});
17:        start_pos ← start_pos + Stage_elt[1]
18:        size ← size − Stage_elt[1]
19:    if size > 0 then
20:        Q[0].Enqueue({start_pos, end_pos, src_id});
21: sync_threads()
22: for i : 0, 1, 2 do
23:    while !isEmpty.Q[i] do
24:        if threadID % Stage_gran[i] == 0 then ▷Only representative thread.
25:            {start_pos, end_pos, src_id} ← Q[i].Dequeue()
26:            Broadcast({start_pos, end_pos, src_id}, Stage_gran[i])
27:        coop for eid in start_pos : end_pos − 1 do
28:            dst_id ← G.edges[eid]
29:            process_edge(src_id, dst_id)    ▷Done by Stage_gran[i] threads.
```

processing the edges in a cyclic fashion (Line 27–29). These queues are managed in shared memory to reduce the overhead of the enqueue and dequeue operations.

E. CPU backend implementation

G2 preserves the high-performance C++ CPU code generation backend from GraphIt. We changed the mid-end analyses and transformations in such a way that they do not affect code generation for CPUs. This was done to avoid compromising performance on CPUs. This is important because some applications like Delta-Stepping perform better on CPUs because of a limited amount of parallelism. We will compare the performance of CPUs vs GPUs for this application in Section VII.

F. Auto-tuning

The G2 compiler exposes a large optimization space, with about 10^6 combinations of different schedules. Even without the hybrid schedules that involve two traversal directions, the compiler can generate up to 288 combinations of schedules for each direction (see Table I). On top of that, integer and floating-point parameters, such as the value of delta for Delta-stepping, blocking size of EdgeBlocking, and thresholds for hybrid schedules, need to be appropriately selected for each input graph and algorithm. Searching through the huge optimization space exhaustively is very time-consuming.

To navigate the schedule space more efficiently, we built an auto-tuner for G2 using OpenTuner [77]. For each direction, the auto-tuner chooses among all 288 combinations of options for load balancing, deduplication, output frontier strategy, blocking, traversal direction, and kernel fusion. For direction-optimized schedules that involve two traversal directions, the auto-tuner combines together two sets of schedules, one for each direction. The auto-tuner converges within 10 minutes on each input graph for most algorithms and produces a schedule that matches the performance of hand-optimized schedules.

Graph Input	Vertex count	Edge count
soc-orkut [78] (OK)	2,997,166	212,698,418
soc-twitter-2010 [78] (TW)	21,297,772	530,051,090
soc-LiveJournal [79] (LJ)	4,847,571	85,702,474
soc-sinaweibo [78] (SW)	58,655,849	522,642,066
hollywood-2009 [79] (HW)	1,139,905	112,751,422
indochina-2004 [79] (IC)	7,414,865	301,969,638
road_usa [80] (RU)	23,947,347	57,708,624
road_central [79] (RC)	14,081,816	33,866,826
roadNet-CA [79] (RN)	1,971,281	5,533,214

TABLE IV: Graph inputs used for evaluation. The edge count shows the number of undirected edges.

VII. Evaluation

In this section, we compare the performance of the code generated from G2's GPU backend with state-of-the-art GPU graph frameworks and libraries on 5 graph algorithms and 9 different graph inputs. We also study the performance tradeoffs and effectiveness of some key optimizations. All of the GPU experiments are performed on an NVIDIA Titan Xp (12 GB memory, 3MB L2 cache, and 30 SMs) and an NVIDIA Volta V100 (32 GB memory, 4MB L2 cache, and 80 SMs). For the CPU performance evaluations, we use a dual-socket Intel Xeon E5-2695 v3 CPUs system with 12 cores per processor, for a total of 24 cores and 48 hyper-threads with 128 GB of DD3-1600 memory and 30 MB last level cache on each socket.

Datasets. We list the input graphs used for our evaluation in Table IV, along with their sizes. These are the same datasets used to evaluate Gunrock [24]. Out of the 9 graphs, soc-orkut, soc-twitter-2010, soc-LiveJournal, soc-sinaweibo, hollywood-2009, and indochina-2004 have power-law degree distributions while road_usa, roadNet-CA, and road_central are road graphs with bounded degree distributions.

Algorithms. We evaluate the performance of G2 and the other frameworks on five algorithms: PageRank (PR), Breadth-First Search (BFS), Delta-Stepping for Single-Source Shortest Paths (SSSP), Connected Components (CC) and Betweenness Centrality (BC). These algorithms evaluate different aspects of the compiler and give insights into how each optimization helps with performance. PR is a topology-driven algorithm that iterates over all edges in every round and is useful in evaluating the performance benefits of the EdgeBlocking optimization. BFS and BC greatly benefit from direction-optimization on graphs with a power-law degree distribution. We use the hybrid scheduling API to get maximum performance for these algorithms. Delta-Stepping makes use of the priority queue API. Both BFS and Delta-Stepping benefit from kernel fusion for high-diameter road graphs. CC uses the algorithm by Soman et al. [38] and benefits from carefully choosing the load balancing strategy.

Comparison Frameworks. We compare G2's performance with three state-of-the-art GPU graph processing frameworks: Gunrock [24], GSWITCH [19], and SEP-Graph [20]. Both

Framework	PR	BFS	Delta-Stepping	CC	BC
Gunrock	2207	2189	1438	3014	1792
GSWITCH	159	164	203	160	280
SEP-Graph	–	481	473	–	–
G2 (algorithm+schedule)	**61**	**66**	**50**	**62**	**128**

TABLE V: Number of lines of code for the five algorithms written using Gunrock, GSWITCH, SEP-Graph, and G2. SEP-Graph does not implement CC BC and PR. The shortest code length for each algorithm is shown in **bold**.

Gunrock and GSWITCH have optimized implementations of BFS for power-law graphs using direction-optimization. SEP-Graph improves the performance of Delta-Stepping and BFS on high-diameter graphs by fusing kernel launches across iterations like in our kernel fusion optimization.

GSWITCH chooses among several optimal parameters for traversal direction, load balancing, and frontier creation using a learned decision tree that makes use of graph characteristics and runtime metrics. Each framework implements all of the algorithms that we evaluate, except for SEP-Graph, which does not implement CC, PR and BC. We did not compare G2 with Groute and IrGL because Groute is outperformed by SEP-Graph [20], and IrGL is not publicly available.

A. Comparison with other Frameworks

Tables VI and VII show the execution times of all of the algorithms in G2 and the other frameworks on the Titan Xp and V100 machines, respectively. G2 outperforms the next fastest of the three frameworks on 66 out of 90 experiments by up to 5.11×. (G2 can be up to 5x slower than the fastest framework in some rare cases) Table V shows the number of lines of code for each algorithm in each framework. G2 always uses significantly fewer lines of code compared to other frameworks.

PR. G2 has the fastest PR on 16 out of 18 experiments. Compared to Gunrock and GSWITCH, G2 is up to 4.2x faster. This is mainly because of the EdgeBlocking optimization that reduces the number of L2 cache misses, as described in Section VI-D. The results are even better on the V100 GPU because of its larger L2 cache size. The graph is divided into fewer subgraphs reducing the iteration overhead.

BFS. G2 has the fastest BFS on 11 of the 18 experiments. G2 outperforms GSWITCH and Gunrock by up to 6.04x and 1.63x, respectively, on the road graphs because Gunrock and GSWITCH do not use the kernel fusion optimization to reduce kernel launch overheads as discussed in Section VI-C. SEP-Graph is only up to 1.24x slower than G2 on the road graphs because it uses asynchronous execution, which is beneficial for high-diameter graphs. However, the better load balancing achieved by ETWC makes G2 faster than SEP-Graph.

On the power-law graphs, direction-optimization is very effective. Both Gunrock and GSWITCH use direction optimization and hence the performance of G2 is very close to both of them. GSWITCH and Gunrock also use idempotent label updates, which introduces a benign race instead of using expensive compare-and-swap. This optimization is specific for BFS and is hard to generalize in a compiler like G2. Even on road graphs, G2 gains significant speedups over GSWITCH and Gunrock because of the kernel fusion optimization as the kernel launch overhead is still the dominating factor.

The indochina-2004 graph is a special case of a power-law graph that does not benefit from direction-optimization because the number of active vertices never crosses our threshold for `PULL` strategy to be beneficial over `PUSH` strategy. Both Gunrock and GSWITCH use direction-optimization for indochina-2004 and suffer from the extra work done in the `PULL` direction. G2 uses the more efficient `PUSH`-only schedule.

CC. For CC, G2 is the fastest on 16 out of the 18 experiments. G2 is up to 3.4x faster than the next fastest framework. All of the frameworks use the same algorithm and execution strategies. We tune the performance by choosing different load balancing strategies for each graph (ETWC for power-law graphs and CM for road graphs). The speedups are not as significant on the Volta GPUs as compared to the Pascal GPUs, because the Volta generation GPUs have semi-warp execution that reduces the importance of load balancing techniques.

Delta-Stepping. G2 has the fastest Delta-Stepping performance on 11 of the 18 graph inputs and runs up to 4.61x faster than the next fastest framework. Delta-Stepping needs the kernel fusion optimization for road graphs because of their high diameter and the low number of vertices processed in each iteration. GSWITCH and Gunrock lack this optimization and thus are slower on road graphs by up to 2.05x and 89x, respectively. On power-law graphs, G2 benefits from the better ETWC load balancing strategy and performs up to 5.11x faster than the next fastest framework. On road-graphs, SEP-Graph executes up to 1.36x times faster because of the highly optimized asynchronous execution. Finally, in some cases, G2 runs more slowly on Volta than on Pascal because the kernel launch overhead and synchronization costs are higher on Volta than on Pascal due to its larger number of SMs (Delta-Stepping does not have enough parallelism to utilize all of the SMs). The other frameworks also have a similar slowdown when moving from Pascal to Volta. Table VIII shows that on road-graphs, G2's CPU implementation is 2x faster than SEP-Graph's. This highlights the need for generating code for both CPU and GPU from the same high-level representation.

BC. G2 has the fastest BC performance on 12 out of 18 experiments. G2 runs up to 2.03x faster than the next fastest framework. Gunrock does not use direction-optimization for BC, which is critical for high performance on power-law graphs. GSWITCH uses direction-optimization, but G2 outperforms GSWITCH because of the better load balancing from ETWC. For high-diameter graphs, G2 benefits greatly from the kernel fusion optimization. Both GSWITCH and Gunrock do not implement kernel fusion.

B. Comparison against CPU

We also compare the performance of G2-generated CPU implementations with G2-generated GPU implementations running on the Titan Xp Pascal generation GPU. G2 generates the same CPU implementation as GraphIt and PriorityGraph [7], [6]. On PR, BFS, and CC, the GPU implementations are faster because these algorithms can easily utilize the large amount of

	PR (time per round)			CC			BC		
Graph	G2	GU	GSW	G2	GU	GSW	G2	GU	GSW
OK	**14.18**	60.63	117.10	**63.31**	71.95	76.85	**26.22**	213.93	37.93
TW	**77.86**	113.55	211.03	**196.78**	374.59	OOM	174.83	505.31	**122.81**
LJ	**7.68**	17.67	OOM	**24.65**	35.96	27.81	28.93	88.04	**26.98**
SW	**102.11**	178.70	338.79	**276.04**	439.51	OOM	**204.32**	1095.60	415.06
HW	**7.01**	22.67	OOM	**12.04**	37.43	18.23	**12.20**	29.03	79.44
IC	18.24	13.16	**9.30**	**31.66**	235.92	43.10	35.78	47.97	**10.70**
RU	**6.32**	10.53	7.62	**20.66**	74.45	31.21	**302.22**	987.93	564.53
RC	**5.56**	9.96	8.86	**27.03**	48.22	27.13	**239.55**	632.97	332.91
RN	**0.43**	0.94	0.47	**1.72**	5.82	3.04	**24.05**	86.44	39.51
	SSSP with Delta-Stepping				BFS				
Graph	G2	GU	GSW	SEP-G	G2	GU	GSW	SEP-G	-
OK	**94.30**	978.44	550.38	434.77	**1.75**	1.92	1.94	6.40	
TW	**114.34**	264.54	233.54	237.12	**21.13**	OOM	22.52	40.18	
LJ	**54.46**	260.19	172.99	172.07	4.66	4.80	**3.68**	11.63	
SW	**685.66**	3470.59	1933.29	2296.01	20.17	OOM	**18.96**	93.69	
HW	**18.44**	74.90	47.54	92.92	**1.89**	2.04	2.30	4.76	
IC	**120.42**	232.55	268.95	511.70	**10.68**	15.92	70.69	55.15	
RU	601.08	53518.87	1238.03	**440.21**	**73.15**	442.19	119.93	84.29	
RC	337.62	29443.51	642.38	**286.03**	**49.59**	OOM	84.98	61.51	
RN	17.01	89.18	27.80	**16.48**	**6.13**	OOM	10.29	6.88	

TABLE VI: Execution time in milliseconds for the 5 algorithms on 9 input graphs for the 4 frameworks in comparison, G2, Gunrock (GU), GSWITCH (GSW), and SEP-Graph (SEP-G) running on an NVIDIA Titan Xp GPU. The fastest result per algorithm-graph combination is shown in **bold**. The PR, BFS, BC, and CC algorithms use unweighted and symmetrized graphs. Single-Source Shortest Path (SSSP) with Delta-Stepping uses the original graphs (not symmetrized) with edge weights. Uniformly random integer weights between 1–1000 are added for soc-orkut, soc-sinaweibo, hollywood-2009, and indochina-2004 because they did not originally have edge weights. OOM indicates that the framework ran out of memory for the particular input.

	PR (time per round)			CC			BC		
Graph	G2	GU	GSW	G2	GU	GSW	G2	GU	GSW
OK	**7.9**	20.59	32.77	**15.18**	-	16.33	**20.05**	213.93	20.56
TW	**39.44**	43.04	103.88	**134.95**	-	137.12	90.69	505.31	**56.44**
LJ	**3.76**	7.66	6.49	**7.9**	-	11.12	21.31	88.04	**16.88**
SW	**50.18**	66.41	163.88	**173.59**	-	290.02	**154.06**	1095.60	216.05
HW	**4.27**	6.47	8.67	7.4	-	**6.85**	11.26	29.03	**6.08**
IC	**13.99**	16.64	46.55	22.31	-	**4.46**	**30.09**	47.97	55.58
RU	3.1	**2.68**	3.22	**12.44**	-	17.94	**330.82**	987.93	536.95
RC	**2.69**	2.61	2.97	**10.01**	-	14.14	**213.39**	632.97	336.78
RN	**0.22**	0.27	0.29	**1.34**	-	2.79	**32.79**	86.44	47.06
	SSSP with Delta-Stepping				BFS				
Graph	G2	GU	GSW	SEP-G	G2	GU	GSW	SEP-G	-
OK	**50.09**	303.15	199.59	164.69	**1.51**	1.66	**1.51**	5.72	
TW	**61.85**	101.33	132.94	117.97	14.73	13.52	**10.60**	38.18	
LJ	**40.05**	131.91	77.95	103.40	**2.68**	3.63	3.05	9.59	
SW	**375.42**	1686.39	1062.56	1066.57	13.86	94.77	**12.26**	70.70	
HW	**18.04**	37.42	26.77	51.75	1.65	1.67	**1.60**	4.81	
IC	100.49	**25.58**	211.85	350.58	**9.03**	12.57	40.60	39.39	
RU	253.11	788.23	390.23	**191.08**	**74.26**	775.16	186.43	97.62	
RC	168.7	429.24	222.05	**128.02**	118.52	434.01	**115.13**	65.49	
RN	21.08	66.79	32.47	**19.46**	10.09	68.35	16.76	**9.33**	

TABLE VII: Execution time in milliseconds for the same experiments and comparison frameworks in Table VI running on an NVIDIA V100 GPU. The fastest result per algorithm-graph combination is shown in **bold**. Gunrock's CC implementation does not support the Volta GPU and has correctness issues.

parallelization and memory bandwidth available on the GPUs. On the other hand, Delta-Stepping, which has less parallelism available when running on road graphs, executes up to 2.07× faster on the CPU due to more powerful cores and larger caches. The execution times for Delta-Stepping on the GPU and CPU can be found in Table VIII. Another advantage of CPUs is that they can process graphs that are much larger than the GPU memory more efficiently. These experiments provide evidence for our claim that neither CPUs nor GPUs are suitable for all algorithms and inputs and shows how a framework that can generate code for both CPUs and GPUs from the same input can help achieve high performance.

C. Performance of ETWC and EdgeBlocking

In this section, we compare the performance of the two optimizations that we introduce in G2, ETWC and EdgeBlocking. To evaluate the performance of the ETWC load balancing scheme, we run the BFS algorithm on all nine graph inputs. We fix the direction to `PUSH` and do not use kernel fusion.

We only choose among the ETWC, TWC, and CM load balancing schemes (TWC and CM are the best performing on social and road graphs, respectively, hence other load balancing schemes are not being compared). The results are shown in Table IX. CM is faster than TWC on graphs that have a regular degree distribution, like road graphs and the

Delta-Stepping for SSSP	G2 CPU	G2 GPU	SEP-Graph
road_usa	**212.30**	601.80	440.21
road_central	**162.49**	337.62	286.03
soc-orkut	106	**94.3**	434.77
soc-LiveJournal	90.05	**54.46**	172.07

TABLE VIII: Comparisons of G2-generated CPU implementations, G2-generated GPU implementations, and SEP-Graph on SSSP with Delta-Stepping. The GPU experiments are run on the Pascal generation GPU. The running times are in milliseconds. The fastest result per graph is shown in **bold**. We did not count the data transfer time from CPU to GPU. We do not show Gunrock and GSWITCH because they are always outperformed by SEP-Graph for SSSP.

Graph	ETWC	TWC	CM
soc-orkut	43.58	**40.69**	42.24
soc-twitter-2010	**106.11**	107.57	116.06
soc-LiveJournal	19.72	20.03	**18.42**
soc-sinaweibo	**226.35**	230.00	230.03
hollywood-2009	**4.94**	5.79	8.17
indochina-2004	**11.38**	11.5	22.16
road_usa	**136.64**	255.89	168.9
road_central	**91.2**	162.54	109.89
roadNet-CA	**13.1**	25.77	16.25

TABLE IX: Execution time (in milliseconds) of BFS (`PUSH` only) using ETWC, TWC, and CM load balancing strategies. The fastest result per graph is shown in **bold**.

indochina-2004 graph, while TWC performs better on social graphs with power-law degree distributions because it is able to separate out vertices of different degrees. TWC is slower on road graphs because of the overhead from load balancing. ETWC outperforms the other two load balancing schemes on 7 of the 9 graphs. ETWC does well both on social and road graphs because it is able to effectively load balance vertices with varying degrees without incurring a large overhead. This is because unlike TWC, ETWC balances the edges only locally within a CTA, thus communicating only using shared memory.

Similarly, we also evaluate the performance gains of using EdgeBlocking by running PR on all of the input graphs. We fix the schedule to use edge-only load balancing (which is the fastest for PR) and compare the execution times with and without EdgeBlocking. The vertex data of these vertices fit in the L2 cache of the GPU. Table X shows the execution time of PR with EdgeBlocking disabled and enabled and the speedup from enabling it. We see that with EdgeBlocking enabled, PR runs up to 2.94x faster. However, EdgeBlocking causes some slowdown on indochina-2004 and hollywood-2009 because of degradation in work-efficiency and the graphs being already somewhat clustered (the IDs of the neighbors of each vertex are in a small range). EdgeBlocking requires us to preprocess the input graphs. Table X shows that the overhead for preprocessing is less than the time required for two rounds of PR and thus can be easily amortized across multiple runs.

Finally, we evaluate the performance impact of kernel fusion. Although kernel fusion is not a new technique presented in this paper, a novelty of the compiler is the algorithm for generating fused kernels for an arbitrary sequence of operations. Table XI shows the execution time of the BFS algorithm with and without kernel fusion on all of the graph inputs on the V100 GPU. We can clearly see that enabling kernel fusion offers up to 3.51x speedup for road graphs, which have a large diameter and run

Graph	Without EB	With EB	Speedup	Preprocessing time
soc-orkut	41.75	**14.18**	2.94x	22.75
soc-twitter-2010	88.25	**77.86**	1.13x	129.43
soc-LiveJournal	15.67	**7.68**	2.04x	20.65
soc-sinaweibo	144.88	**102.11**	1.41x	141.86
hollywood-2009	**7.01**	7.02	0.99x	12.00
indochina-2004	**18.24**	19.55	0.93x	32.25
road_usa	170.75	**126.41**	1.35x	11.77
road_central	167.82	**111.37**	1.50x	8.20
roadNet-CA	8.93	**8.74**	1.02x	1.08

TABLE X: Execution time and preprocessing time (in milliseconds) per round of PR with and without EdgeBlocking. The Speedup shows improvement in execution time with EdgeBlocking enabled.

Graph	Without kernel fusion	With kernel fusion	Speedup
soc-orkut	**1.51**	2.21	0.68x
soc-twitter-2010	**14.73**	21.42	0.69x
soc-LiveJournal	**2.68**	5.24	0.51x
soc-sinaweibo	**13.86**	18.56	0.74x
hollywood-2009	**1.65**	2.51	0.65x
indochina-2004	**9.03**	14.26	0.63x
road_usa	261.28	**74.26**	3.51x
road_central	161.2	**118.52**	1.36x
roadNet-CA	22.85	**10.09**	2.26x

TABLE XI: Execution time (in milliseconds) for BFS with and without kernel fusion. The last column shows the speedup with kernel fusion enabled.

for iterations with very few vertices to process per iteration. On the other hand, enabling kernel fusion slows down the execution for power-law degree graphs by up to 2x because it greatly affects load balancing, which is critical for power-law degree graphs. Fused kernels also increase register pressure, which affects kernels that process a lot of vertices per iteration. These experiments show that it is critical to tune the kernel launch decision using the scheduling language.

VIII. Conclusion

We introduce the G2 GraphIt GPU compiler for writing high-performance graph algorithms for both CPUs and GPUs. G2 also introduces a novel GPU scheduling language and allows users to search through many different combinations of load balancing, traversal direction, active vertexset management, and work-efficiency optimizations. We propose two performance optimizations, ETWC and EdgeBlocking, to improve load balancing and locality of edge processing, respectively. We evaluate G2 on 5 algorithms and 9 graphs and show that it achieves up to 5.11× speedup over the next fastest framework, and is the fastest on 66 out of the 90 experiments.

IX. Acknowledgments

This research was supported by the MIT Research Support Committee Award, DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, DARPA D3M Award #FA8750-17-2-0126, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, and Google Faculty Research Award.

REFERENCES

[1] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," in *Proceedings of the 2018 World Wide Web Conference (WWW)*, 2018, pp. 1775–1784.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2018, pp. 974–983.

[3] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: Real-time content recommendations at Twitter," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1281–1292, Sep. 2016.

[4] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," in *USENIX Annual Technical Conference (USENIX ATC)*, 2013, pp. 49–60.

[5] S. Pallottino and M. G. Scutellà, *Shortest path algorithms in transportation models: Classical and innovative aspects*, 1998, pp. 245–281.

[6] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 121:1–121:30, 2018.

[7] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with GraphIt," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2020, p. 158170.

[8] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2017, pp. 16:1–16:10.

[9] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[10] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *IEEE Data Compression Conference (DCC)*, 2015, pp. 403–412.

[11] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018, pp. 246–260.

[12] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[13] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren, "Graphphi: Efficient parallel graph processing on emerging throughput-oriented architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[14] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.

[15] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 456–471.

[16] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable GPU graph traversal," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 14:1–14:30, Feb. 2015.

[17] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 1–19.

[18] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017, pp. 235–248.

[19] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on GPUs," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 201–213.

[20] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "SEP-Graph: Finding shortest execution paths for graph processing under a hybrid framework on GPU," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 38–52.

[21] H. Liu and H. H. Huang, "SIMD-x: Programming and processing of graph algorithms on gpus," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 411–428.

[22] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014, pp. 239–252.

[23] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 39–50.

[24] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, p. 3, 2017.

[25] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Multigraph: Efficient graph processing on GPUs," in *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 27–40.

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

[27] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 622–636.

[28] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-GPU-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. New York, NY, USA: Association for Computing Machinery, 2020.

[29] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the GraphBLAS," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.

[30] C. Yang, A. Bulucc, and J. D. Owens, "Implementing push-pull efficiently in GraphBLAS," in *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, 2018, pp. 89:1–89:11.

[31] U. Meyer and P. Sanders, "δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[32] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A compiler for graph analytics on distributed, heterogeneous platforms," in *European Conference on Parallel Processing (Euro-Par)*, 2018, pp. 249–264.

[33] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *International Conference on High-Performance Computing (HiPC)*, 2007, pp. 197–208.

[34] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 267–276.

[35] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 1–12.

[36] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin, "Frog: Asynchronous graph processing on GPU with hybrid coloring model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 1, pp. 29–42, 2017.

[37] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[38] J. Soman, K. Kishore, and P. Narayanan, "A fast GPU algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[39] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 463–474.

[40] S. Che, "GasCL: A vertex-centric graph model for GPUs," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[41] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to GPUs," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016, pp. 447–461.

[42] A. Gaihre, Z. Wu, F. Yao, and H. Liu, "XBFS: eXploring runtime optimizations for breadth-first search on GPUs," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019, pp. 121–131.

[43] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 233–245.

[44] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, p. 601614.

[45] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015, pp. 183–193.

[46] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 349–362.

[47] M. S. Lam, S. Guo, and J. Seo, "SociaLite: Datalog extensions for efficient social network analysis," in *IEEE International Conference on Data Engineering (ICDE)*, 2013, pp. 278–289.

[48] C. R. Aberger, A. Lamb, S. Tu, A. Ne, K. Olukotun, and C. Re, "EmptyHeaded: A relational engine for graph processing," vol. 42, no. 4, Oct. 2017, pp. 20:1–20:44.

[49] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 237–251.

[50] 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)*, vol. 5, no. 3, pp. 13:1–13:39, 2018.

[51] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 301–316.

[52] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 752–768.

[53] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H.-V. Dang, and M. Snir, "Gluon-Async: A bulk-asynchronous system for distributed and heterogeneous graph analytics," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 15–28.

[54] 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 Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, Oct. 2015.

[55] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010, pp. 340–349.

[56] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[57] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[58] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[59] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 31–46.

[60] 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 (SOSP)*, 2013, pp. 472–488.

[61] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference (ATC)*, 2016, pp. 507–522.

[62] K. Wang, G. Xu, Z. Su, and Y. D. Liu, "GraphQ: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single PC," in *USENIX Annual Technical Conference (ATC)*, 2015, pp. 387–401.

[63] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," pp. 608–621, 2018.

[64] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li, "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proceedings of the Fourteenth European Conference on Computer Systems (EuroSys)*, 2019, p. 38.

[65] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017, pp. 527–543.

[66] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "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)*, 2017, pp. 389–404.

[67] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015, pp. 375–386.

[68] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity SSDs," in *USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 45–58.

[69] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "RStream: marrying relational algebra with streaming for efficient graph mining on a single machine," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 763–782.

[70] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.

[71] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 820–831.

[72] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with Milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*, 2016, pp. 299–312.

[73] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Cache-aware sparse matrix formats for Kepler GPU," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2014, pp. 281–288.

[74] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 300–314.

[75] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 12:1–12:10.

[76] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2017, pp. 93–104.

[77] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[78] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[79] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

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