Compiling Graph Applications for GPUs with GraphIt

Ajay Brahmkshatriya  
MIT CSAIL  
USA  
ajaybr@mit.edu

Yunming Zhang  
MIT CSAIL  
USA  
yunning@mit.edu

Changwan Hong  
MIT CSAIL  
USA  
changwan@mit.edu

Shoaib Kamil  
Adobe Research  
USA  
kamil@adobe.com

Julian Shun  
MIT CSAIL  
USA  
jshun@mit.edu

Saman Amarasinghe  
MIT CSAIL  
USA  
saman@csail.mit.edu

Yunming Zhang  
MIT CSAIL  
USA  
yunning@mit.edu

Julian Shun  
MIT CSAIL  
USA  
jshun@mit.edu

Saman Amarasinghe  
MIT CSAIL  
USA  
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 reimplementing the entire algorithm from scratch in a low-level language 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 Warsps 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, GPUs, 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 critical because these applications often need to process very large graphs and/or have strict latency requirements [1].
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 G2 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, EdgeBlocking 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 code that is exactly same as 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.

IroGL [17] is another compiler framework that introduces a programming language and an intermediate representation specifically for graph applications on GPUs. IroGL first proposed the kernel fusion optimization, which reduces the overhead of kernel launch for SSSP and BFS on high-diameter road networks. However, IroGL supports a much narrower space of optimizations (see Table I). Unlike G2, IroGL does not support code generation for CPUs and GPUs from the same input.
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 critical for 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 (see 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. G2 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) directions are needed for achieving high performance in many graph algorithms. These two directions have different memory access patterns and parallelism, and must be optimized separately.
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 program.

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.

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 vertexset 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, and not just 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 threads and inserts atomics.

Apart from inserting atomics, the compiler also transforms the UDF to enqueue the vertices into the output frontier when their vertex data is updated. The compiler refers to the schedule and the tracking variable to select the enqueue function to use.
**TABLE II. DESCRIPTION OF THE SIMPLEGPUSCHEDULING TYPE AND ASSOCIATED \texttt{config} FUNCTIONS. THE DEFAULT VALUE FOR EACH PARAMETER IS IN BOLD.**

<table>
<thead>
<tr>
<th>Config function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleGPUSchedule</td>
<td>SimpleGPUSchedule s0</td>
<td>Create a new SimpleGPUSchedule object. Optional copy constructor argument.</td>
</tr>
</tbody>
</table>

**Scheduling functions for \texttt{edges.apply}**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
</table>
| configLoadBalance        | load_balance_type,          | Configure the load balancing scheme to be used from one of VERTEX_BASED, CM, enable_fusion
|                          | blocking_type (optional),   | Option values are CM, WM, STRICT, EDGE_ONLY, ETWC, and TMC. Optionally enable_fusion
|                          | int32_t blocking_size      | enable_fusion                                                                 |
| configDirection          | direction, front_rep        | Configure the direction of updates of vertex data between PUSH and FULL. Optionally enable_fusion
|                          | (optional)                 | enable_fusion                                                                 |
| configFrontierCreation   | frontiers_type              | Configure how the output frontier will be created. Options are FUSED, UNFUSED_BITMAP, enable_fusion enable_fusion
|                          |                             | enable_fusion                                                                 |
| configDeduplication      | dedup_enable,               | Configure if kernel fusion is enable_fusion enable_fusion, or disable_fusion. enable_fusion
|                          | dedup_type                  | enable_fusion                                                                 |

**Scheduling functions for \texttt{edges.applyUpdatePriority}**

<table>
<thead>
<tr>
<th>Function</th>
<th>parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| configKernels            | int & delta                | Configure the delta value to be used when creating the buckets in the priority queue. enable_fusion

**Scheduling functions for \texttt{while(condition)}**

<table>
<thead>
<tr>
<th>Function</th>
<th>parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| configKernelFunction     | enable_fusion              | Configure if kernel fusion is enable_fusion enable_fusion, or disable_fusion. enable_fusion

**TABLE III. DESCRIPTION OF THE HYBRIDGPUSCHEDULING TYPE AND CONSTRUCTOR. DEFAULT VALUE FOR EACH PARAMETER IS SHOWN IN BOLD.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| HybridGPUSchedule        | hybrid_criteria,           | Create a HybridGPUSchedule object by combining two SimpleGPUSchedule objects with a runtime condition (currently only \texttt{INPUT_VERTEXSET_SIZE} and a threshold. enable_fusion
|                          | float threshold,           | enable_fusion                                                                 |
|                          | SimpleGPUSchedule s1,      | enable_fusion                                                                 |
|                          | SimpleGPUSchedule s2       | enable_fusion                                                                 |

**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.**

**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 \texttt{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 \texttt{__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 \texttt{__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 implementation of the load-balancing library that allows the core library to be instantiated for different load balancing schemes.

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 2.94x, 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(N/segmentSize)
4: segmentSize ← segmentSize[segment] + 1
5: Gout ← new Graph
6: for e : G.edges do
7:   segment ← floor(e.dst/N)
8:   segmentSize[segment] ← segmentSize[segment] + 1
9:   prefixSum ← prefix_sum(segmentSize)
10: for e : G.edges do
11:   index ← prefixSum[segment]
12:   index ← index + e
13:   Gout[edges[index]] ← e
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
3:   start_vertex ← G.N * segIdx
4:   end_vertex ← G.N * (segIdx + 1)
5:   coop for e in G.edges : segIdx do
6:     e ← G.edges[segIdx]
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 divergence prone individual thread stage.

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 processing the edges in a cyclic fashion (Line 27–29). These
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);
7:   start_pos ← G.rowptr[src_id]
10: if Stage_elts[2] > 0 then
11:   Q[2].Enqueue({start_pos, start_pos + Stage_elts[2], src_id});
12:   start_pos ← start_pos + Stage_elts[2]
13:   size ← size - Stage_elts[2]
14:   Stage_elts[1] ← [size/Stage_gran[1]] × Stage_gran[1] (WarpSize).
15: if Stage_elts[1] > 0 then
16:   Q[1].Enqueue({start_pos, start_pos + Stage_elts[1], src_id});
17:   start_pos ← start_pos + Stage_elts[1]
18:   size ← size - Stage_elts[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 !input_empty.Q[i][1] do
24:     if threadID % Stage_gran[1] == 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[1])
27:     coop for edge in start_pos + Stage_elts[1] - 1 do
28:     dst_id ← G.edges[ed]
29:   process_edge(src_id, dst_id) done by Stage_gran[1][1] threads.

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.

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 GPU 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-orcut, 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), Breath-Frist 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 right 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 Gunrock and GSWITCH have optimized implementations of
**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 5.11x. 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. GSWITCH 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.

**BC.** G2 has the fastest BC 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 parallelization and memory bandwidth available on the GPUs. On the other hand, **Delta-Stepping**, which has less parallelism...
The execution times for Delta-Stepping on the GPU and CPU are shown in Table VI. 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-orokut, 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.

### Table VI: Execution time in milliseconds for the same experiments and comparison frameworks in Table VI running on an NVIDIA V100 GPU.

<table>
<thead>
<tr>
<th>PR (time per round)</th>
<th>CC</th>
<th>BC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>G2</td>
<td>GU</td>
</tr>
<tr>
<td>OK</td>
<td>14.18</td>
<td>60.63</td>
</tr>
<tr>
<td>TW</td>
<td>77.86</td>
<td>113.55</td>
</tr>
<tr>
<td>LJ</td>
<td>7.68</td>
<td>17.67</td>
</tr>
<tr>
<td>SW</td>
<td>102.11</td>
<td>178.70</td>
</tr>
<tr>
<td>HW</td>
<td>7.01</td>
<td>22.61</td>
</tr>
<tr>
<td>IC</td>
<td>18.24</td>
<td>13.16</td>
</tr>
<tr>
<td>RU</td>
<td>6.32</td>
<td>10.53</td>
</tr>
<tr>
<td>RC</td>
<td>5.56</td>
<td>9.96</td>
</tr>
<tr>
<td>RN</td>
<td>0.43</td>
<td>0.94</td>
</tr>
</tbody>
</table>

#### Table VII: 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-orokut, 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.

<table>
<thead>
<tr>
<th>PR (time per round)</th>
<th>CC</th>
<th>BC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>G2</td>
<td>GU</td>
</tr>
<tr>
<td>OK</td>
<td>94.30</td>
<td>978.44</td>
</tr>
<tr>
<td>TW</td>
<td>114.34</td>
<td>264.54</td>
</tr>
<tr>
<td>LJ</td>
<td>54.46</td>
<td>264.54</td>
</tr>
<tr>
<td>SW</td>
<td>685.60</td>
<td>322.76</td>
</tr>
<tr>
<td>HW</td>
<td>18.44</td>
<td>74.90</td>
</tr>
<tr>
<td>RC</td>
<td>120.42</td>
<td>232.55</td>
</tr>
<tr>
<td>RU</td>
<td>601.08</td>
<td>53518.87</td>
</tr>
<tr>
<td>RN</td>
<td>337.62</td>
<td>29443.51</td>
</tr>
</tbody>
</table>

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, and 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 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.
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 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 significantly 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 $5.11 \times$ speedup over the next fastest framework, and is the fastest on 66 out of the 90 experiments.

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


