

# Custom Code Generation for a Graph DSL

Bikash Gogoi<sup>1</sup>, Unnikrishnan Cheramangalath<sup>2</sup>, Rupesh Nasre<sup>3</sup>

<sup>1</sup> Department of CSE, Indian Institute of Technology, Madras,  
bikashgogoi001@gmail.com

<sup>2</sup> Singapore University of Technology and Design, Singapore,  
unnikrishnan\_cheramangalath@sutd.edu.sg

<sup>3</sup> Department of CSE, Indian Institute of Technology, Madras, rupesh@iitm.ac.in

**Abstract.** We present challenges faced in making a domain-specific language (DSL) for graph algorithms adapt to varying requirements to generate a spectrum of efficient parallel codes. Graph algorithms are at the heart of several applications, and achieving high performance with them has become critical due to the tremendous growth of irregular data. However, irregular algorithms are quite challenging to parallelize automatically, due to access patterns influenced by the input graph – which is unavailable until execution. Former research has addressed this issue by designing DSLs for graph algorithms, which restrict generality but allow efficient code-generation for various backends. Such DSLs are, however, too rigid, and do not adapt to changes in backends or to input graph properties or to both. We narrate our experiences in making an existing DSL, named Falcon, adaptive. The biggest challenge in the process is to not change the DSL code for specifying the algorithm. We illustrate the effectiveness of our proposal by auto-generating codes for vertex-based versus edge-based graph processing, synchronous versus asynchronous execution, and CPU versus GPU backends from the same specification.

## 1 Introduction

Graphs model several real-world phenomena such as friendship, molecular interaction and co-authorship. Several graph algorithms have been designed across domains to compute such relationships between entities. Performance of these graph algorithms has become critical today due to the explosive growth of unstructured data. For instance, to simulate a simple physical phenomenon, an algorithm may have to work with billions of particles.

On the other side, computer hardware is witnessing rapid changes with new architectural innovations. Exploiting these architectures demands complex coding and good compiler support. The demand intensifies in the presence of parallelization. It is not uncommon to see a twenty-line textbook graph algorithm implemented using several hundred lines of optimized parallel code.

It would be ideal if a graph algorithm can be programmed at a high-level without worrying about the nuances of the hardware. Domain-specific languages (DSLs) for graph data analytics allow programmers to write complex algorithmic codes with minimal hardware knowledge and less programming effort. The

code generator of the DSL compiler emits efficient parallel code [8,20,2]. Such DSLs often disallow writing arbitrary programs, trading off generality for performance. This makes programming parallel hardware easy, and adapting to changes manageable.

In this work, we focus on a recent graph DSL named Falcon [2,1], which supports a wide variety of backends: CPU, GPU, multi-GPU, and distributed systems with CPU and GPU. It extends C language to allow graph processing being specified at a high-level. Falcon provides special data types (such as `worklist` and `set`) as well as constructs (such as `foreach` and `reduction`) for simplifying algorithm specification and aiding efficient code generation. Table 1 compares various graph processing frameworks.

Graph algorithms are challenging to parallelize due to their inherent *irregularity*, which makes their data-access, control-flow, and communication patterns data-dependent. For instance, vertex-based processing works well for road networks, but social networks demand an edge-based processing. Social networks have a skewed degree distribution and road networks have a large diameter. Sequential processing of parallel loops demands synchronous execution, but independent loops can be more efficient with asynchronous processing. Dense subgraphs can be efficiently processed using a topology-driven approach [16], whereas a data-driven worklist-based approach is better suited for sparse subgraphs. Similarly, backend optimizations are quite different for different targets such as CPU and GPU.

Falcon, and other graph DSLs, allow writing code for a particular kind of processing. The code written in Falcon DSL needs modifications for an alternative way of processing. Various syntactic elements in the program need to be changed for the alternative way. Thus, the code needs to be written separately for vertex-based and edge-based processing, for instance. It would be ideal if one could generate different kind of code from the *same DSL specification*. Such a setup greatly simplifies the algorithmic specification, and also allows generating code for various situations / backends / graph types from the same specification.

In this paper, we highlight the challenges faced in building such a versatile compiler. In particular, we make the following contributions:

- We present a compiler which generates different implementations for the *same DSL program* for a graph algorithm. In particular, the compiler can generate vertex-based or edge-based processing, synchronous or asynchronous

Tool	DSL	Hardware Support			Iterators		
		CPU	GPU	multi-GPU	Edge	Vertex	Worklist
GreenMarl [9]	✓	✓	×	×	✓	✓	✓
Galois [18]	×	✓	×	×	×	×	✓
<b>Falcon</b> [2]	✓	✓	✓	✓	✓	✓	✓
Totem [6]	×	✓	✓	✓	×	×	×
Gunrock [24]	×	×	✓	×	×	×	✓
LonestarGPU [17]	×	×	✓	×	✓	✓	✓

Table 1: Comparison of different graph frameworks

codes, topology-driven versus data-driven processing, and CPU or GPU or multi-GPU codes.

- We illustrate the effectiveness of the proposed compiler using several graph algorithms and several graphs of various types. The performance of the code generated with the proposed compiler is compared against other hand-tuned as well as generated codes.

## 2 Falcon

Falcon [2] is a Graph DSL which supports CPU, GPU and multi-GPU machines. It supports various data types, parallelization and synchronization constructs, and reduction operations. This makes programming graph analytic algorithms easy for heterogeneous targets. Falcon also supports dynamic graph algorithms.

Falcon Graph DSL has data types `Graph`, `Point`, `Edge`, `Set` and `Collection`. `Graph` stores a graph object, which consist of points and edges. Each `Point` is stored as a union of `int` and `float`. `Edge` consists of source and destination points, and *weight*. `Set` is a static collection and implemented as a *Union-Find* data structure. The `Collection` data type is dynamic and its size can vary at runtime. Elements can be added to and deleted from a collection object at runtime.

The `foreach` statement is the parallelization construct of Falcon. It can be used to iterate in different ways on different elements of graph object as shown in Table 2. `Parallel Sections` statement of Falcon is used to write programs which use multiple devices of a machine at the same time. Falcon also supports reduction operations such as add (*RADD*) and mul (*RMUL*). It has atomic library functions *MIN*, *MAX* etc., which are necessary for graph algorithms as they are *irregular*. The synchronization primitive of Falcon DSL is `single` statement. It is a non-blocking lock and can be used to lock one element or a collection of elements, as shown in Table 3.

Data Type	Iterator	Description
<code>Graph</code>	<code>points</code>	Iterate over all points in graph
<code>Graph</code>	<code>edges</code>	Iterate over all edges in graph
<code>Point</code>	<code>nbrs</code>	Iterate over all neighboring points (Undirected Graph)
<code>Point</code>	<code>innbrs</code>	Iterate over all src point of incoming edges
<code>Point</code>	<code>outnbrs</code>	Iterate over dst point of outgoing edges
<code>Set</code>		Iterate over all items in a Set
<code>Collection</code>		Iterate over all items in a Collection

Table 2: `foreach` statement iterators in Falcon

<code>single(t1) {stmt block1} else {stmt block2}</code>	The thread that gets a lock on item t1 executes stmt block1 and other threads execute stmt block2.
<code>single(coll) {stmt block1} else {stmt block2}</code>	The thread that gets a lock on all elements in the collection executes stmt block1 and others execute stmt block2.

Table 3: `single` statement (synchronization) in Falcon

---

**Algorithm 1:** SSSP: iterating over Points in Falcon

---

```
1 int changed = 0; // Global variable
2 relaxgraph(Point p, Graph graph) {
3     foreach (t In p.outnbrs)
4         MIN(t.dist, p.dist + graph.getweight(p, t), changed);
5 }
6 main(int argc, char *argv[]) {
7     Graph graph;
8     graph.addPointProperty(dist, int);
9     graph.read(argv[1]);
10    //make dist infinity for all points.
11    foreach (t In graph.points)t.dist=MAX_INT;
12    graph.points[0].dist = 0; // source has dist 0
13    while( (1) ){
14        changed = 0;
15        foreach (t In graph.points) relaxgraph(t,graph);
16        if (changed == 0) break; //reached fix point
17    }
18 }
```

---

A graph object can be processed in multiple ways in Falcon. This leads to the flexibility of the same algorithm being specified in different ways. A programmer can iterate over *edges* of a graph object and then extract the source (*src*) and the destination (*dst*) points of each edge. Another method is to iterate over all *points* of the graph object. Then for each point, the processing can iterate over *outnbrs* or *innbrs*. This is illustrated in Algorithms 1 and 2.

---

**Algorithm 2:** SSSP: iterating over Edges in Falcon

---

```
1 int changed = 0; // Global variable
2 relaxgraph(Edges e, Graph graph) {
3     Point (graph)p,(graph)t;
4     p=e.src;
5     t=e.dst;
6     MIN(t.dist, p.dist + e.weight, changed);
7 }
8 main(int argc, char *argv[]) {
9     Graph graph;
10    graph.addPointProperty(dist, int);
11    graph.read(argv[1]);
12    //make dist infinity for all points.
13    foreach (t In graph.points)t.dist=MAX_INT;
14    graph.points[0].dist = 0; // source has dist 0
15    while( (1) ){
16        changed = 0;
17        foreach (e In graph.edges) relaxgraph(e,graph);
18        if (changed == 0) break; //reached fix point
19    }
20 }
```

---

Both the algorithms are for Single Source Shortest Path (SSSP) computation. It computes the shortest path from source point (point zero) to all other points in the graph object. In Algorithm 1, the processing is done using *points* (Line 15) and *outnbrs* (Line 4) iterators. In Algorithm 2, the computation is performed using *edges* (Line 17) iterator. In both the algorithms all the edges  $t \rightarrow p$  in the graph object are considered. Then *dist* value of point *t* is reduced

to  $\text{Min}(t.\text{dist}, p.\text{dist} + \text{weight}(p \rightarrow t))$  using the atomic function *MIN*. If there is any change in the value of *t.dist*, the variable *changed* is set to one. The computation stops when the value of *dist* does not change for any point in the graph object. Performance of an algorithm depends on the graph structure, hardware architecture, etc. Algorithm 1 may perform well over Algorithm 2 for one input graph, but may not for another, on the same hardware architecture. This depends upon several properties such as the degree of vertices and diameter of the graph object etc. Similarly the worklist (*Collection*) based code also needs to be coded separately in Falcon.

Such a flexible processing is an artifact of *irregular* algorithms (such as graph algorithms) wherein the data-access pattern, the control-flow pattern, as well as the communication pattern is unknown at compile time, as all are dependent on the graph input. Thus, it is difficult to identify which method would be suitable for an algorithm: it depends on the graph object.

The random graphs (Erdős Rényi model) typically perform well with iterating over *points*. The social and rmat graphs which follow power-law degree distribution [6] are benefited mostly by iterating over *edges*, especially on GPU devices. Power-law degree distribution indicates huge variance in degree distribution of the vertices. This can result in thread-divergence in GPU, when parallelized over *points* and iterated over their *outnbrs* or *innbrs*. Road networks benefits with worklist based processing on CPU.

Our goal in this work is to bridge the gap between easy DSL specification and versatility in generating various kinds of codes. Thus, from the same Falcon specification, we want to generate vertex-based or edge-based OpenMP or CUDA codes.

### 3 Our Approach

Algorithm 3 shows the Falcon DSL code for Breadth First Search (BFS) computation. The algorithm is vertex-based and uses *points* (graph nodes) and *outnbrs* (of each node) iterators. The above Falcon program is for CPU. A programmer has to write separate programs for GPU which will have *<GPU>* in the DSL code [2]. Similarly, different programs need to be written for edge-based or worklist-based codes. It would be ideal if the programmer simply specifies *what* rather than *how*, and the DSL compiler takes care of the appropriate code generation. We support it in this work. In our proposal, the programmer needs to specify simply different compile-time arguments. This triggers generation of parallel C++ (CUDA) code matching the output of the *edge*, *vertex* or *worklist* based BFS, targeting CPU (GPU respectively), from a single DSL code. We explain each of these transformations in the subsections below.

#### 3.1 Vertex-based versus Edge-based

Edge-based processing improves load-balance, while vertex-based codes improve propagation of information across the graph and can also reduce synchronization

---

**Algorithm 3:** BFS Algorithm in Falcon for CPU

---

```
1 int changed = 0, lev = 0;
2 void BFS(Point p, Graph graph, int lev) {
3     foreach( t In p.outnbrs ){
4         if( t.dist > lev + 1 ){
5             t.dist = lev + 1;
6             changed = 1;
7         }
8     }
9 }
10 int main(int argc, char *argv[]) {
11     Graph graph;
12     graph.addPointProperty(dist, int);
13     graph.read(argv[3]);
14     foreach(t In graph.points) t.dist=1234567890;
15     graph.points[0].dist = 0;
16     while( 1 ){
17         changed = 0;
18         foreach(t In graph.points)(t.dist == lev) BFS(t, graph, lev);
19         if (changed == 0) break;
20         lev++;
21     }
22 }
```

---

requirements. Conversion of vertex-based to edge-based and vice-versa are done completely at the abstract-syntax tree (AST) level by traversing the AST and modifying its eligible parts. An important conversion non-triviality stems from the edge-based processing being a single loop, while the corresponding vertex-based processing is a nested loop (outer loop over vertices, and inner loop over all the neighbors of each vertex). Pseudo-code for the vertex-based to edge-based transformation is presented in Algorithm 4. The other way is similar.

In vertex-based to edge-based conversion, the subtree is eligible for conversion if: (i) the subtree is rooted at a function node whose only child is a node for a `foreach` which iterates through a point's neighbors, and (ii) the function is the only statement in the body of a `foreach` which iterates through the graph-points. Once the eligible parts are found, we switch the points iterator of the `foreach` statement from which the function is called to edge iterator, and then remove the `foreach` statement in the function. The conversion also requires change in the function's signature as its argument was earlier a point, while now it is an edge. It also necessitates defining two new variables at the beginning of the function corresponding to the source and the destination of the edge. The name of one of the two variables is the name of the point which was the former parameter of the function. The other variable's name is derived from the iterator of the `foreach` statement removed from the function earlier. The order in which these names are mapped to the variables depends on the iterator used

---

**Algorithm 4:** Code transformation for vertex based code

---

```
Input: Falcon vertex-based DSL code
Output: C++/CUDA edge-based code based on the command-line
1 begin Step1:- Mark outermost foreach statement (Done by Falcon Parser).
2 if statement.type == foreach level == 0 then t.outer = true ;
3 if statement.type == foreach level == 1 then t.outer = false ;
4 end Step1
5 begin Step2:- Convert vertex code to edge code
6 forall foreach statement f1 in program do
7   if f1.outer == true level f1.iterator == points then
8     forall foreach statement f2 in program do
9       if f2.def.fun == f1.call.fun level f2.itr == outbrs || f2.itr == innbrs then
10        modify iterator of f1 to edges
11        modify 1st argument to Edge in f2.def.fun
12        create f2.itr and f1.itr in f2.def.fun using Edge
13        remove foreach in f2.def.fun
14        // generate code (Done by Falcon)
15      end
16    end
17  end
18 end
19 end Step2
```

---

in the removed foreach. If the iterator is over out-neighbors, the name of the iterator is mapped to the destination vertex of the edge. Otherwise, we map it to the source vertex. Such an implementation allows the rest of the processing in the iteration to be arbitrary, and reduces the number of alterations the compiler needs to perform to the underlying code.

An important artifact of this processing conversion is that it affects the way graph is stored in memory. In vertex-based code, Falcon (and other frameworks) store the graph in compressed sparse-row (CSR) format. Compared to edge-list representation, CSR format reduces the storage requirement and allows quick access to a vertex's out-neighbors. In edge-based codes, on the other hand, the graph is stored in edge-list format (source destination weight) which enables quick retrieval of the source and the destination points of an edge.

### 3.2 Synchronous versus Asynchronous

By default, Falcon generates BSP-style synchronous code, that is, it inserts a barrier at the end of a parallel construct. While this works well in several codes and eases arguing about the correctness (due to data-races restricted to within-iteration processing across threads), synchronous processing may be overly prohibitive in certain contexts. Especially, in cases where processing across iterations is independent and the hardware does not necessarily demand single-instruction multiple data (SIMD) execution, asynchronous processing may improve performance. Arguing about the correctness-guarantees gets so involved with asynchrony, that some DSLs enforce synchronous-only code generation.

Our proposal is to allow the programmer to generate synchronous or asynchronous code without having to change the algorithm specification code in the DSL. Achieving this necessitates identifying independent processing in the code.

---

**Algorithm 5:** Driver code to generate asynchronous code

---

**Input:** CFG of the function where kernels are launched.  
**Output:** None

```
1 foreach( Node nd in cfg ){  
2   nd.visited = 0  
3   nd.barrier = False  
4   nd.predecessor_count = 0  
5 }  
6 BFS_Mod(cfg.root)// set predecessor_count of nodes by running bfs from root of the CFG  
7 parallelize(cfg.root, None) // Algorithm 6
```

---

Towards this, we maintain read and write sets of global variables and the graph attributes used in each *target function* separately. A *target function* is a function which is being called in the body of a `foreach` statement and the function call is the only statement inside the body of the `foreach`. On CPU, it is the parallel loop body, while on GPU, this function becomes the kernel.

The code conversion has two steps, as shown in Algorithms 5 and 6. In Step 1, we mark nodes in the control-flow graph (CFG); and in Step 2, we generate the appropriate code. In Step 1, we construct the CFG of the *target function* call. Using the read and the write sets corresponding to each of the target functions, we mark each node of the CFG as *barrier-free* or not. A barrier-free node signifies that the target function corresponding to the barrier-free node can be executed concurrently with the children of this node. A node is barrier-free if: (i) there is no dependency between the node and each of its children in CFG. (ii) there is no dependency between the node and the codes between the node and its children.

If a node is *barrier-free*, we pass the read and the write sets of the node to its children. We do this so that the grand-child should not have any dependency with the grand-parent node to declare its parent *barrier-free* (and so on). We follow this process to mark all the CFG nodes in breadth-first search order.

In Step 2, based on the target code the programmer wants, different procedures are followed to make the code asynchronous. If the target is GPU, all the nodes marked as *barrier-free* do not contain a barrier `cudaDeviceSynchronize()` after the kernel launch. Also, each of the *barrier-free* kernels is launched in different streams of the same GPU. On the other hand, if the target is CPU, the *target function* call corresponding to the *barrier-free* node is put in a section of an OpenMP parallel region, and its children and the code between the node and its children in another section. The compiler then recursively checks if the child nodes are barrier-free or not. If they are, then a new OpenMP `parallel` sections construct is introduced inside the section where the child was put in earlier, because of its *barrier-free* parent node. This recursive introduction of parallel sections continues until a *non-barrier-free* node is found, or until the processing reaches the end of the function where these *target functions* are called. The introduction of OpenMP constructs is done by adding new nodes in the AST. For a parallel region, two nodes are added: one each for the start and the end of the construct. In a similar manner, for each section, a node for the start and another node for the end is added. Adding these nodes is easy if both the

---

**Algorithm 6:** Mark node as barrier/barrier-free

---

```
1 parallelize(Node node, Node knode) {
2   if node.stmt.type == KERNEL_LAUNCH then
3     rset = get_read_set_from_function(node.stmt.function)
4     wset = get_write_set_from_function(node.stmt.function)
5     if knode != None then
6       if  $rset \cap knode.wset \neq \text{EMPTY}$  ||  $wset \cap knode.rset \neq \text{EMPTY}$  ||  $wset \cap$ 
7         |  $knode.wset \neq \text{EMPTY}$  then
8         |   knode.barrier = True
9       else
10        | node.rset = node.rset  $\cup$  knode.rset
11        | node.wset = node.wset  $\cup$  knode.wset
12      end
13    end
14    knode = node
15  else
16    if knode != None then
17      rset = get_read_set_from_statement(node.stmt)
18      wset = get_write_set_from_statement(node.stmt)
19      if  $rset \cap knode.wset \neq \text{EMPTY}$  ||  $wset \cap knode.rset \neq \text{EMPTY}$  ||  $wset \cap$ 
20        |  $knode.wset \neq \text{EMPTY}$  then
21        |   knode.barrier = True
22        |   knode = None
23      end
24    end
25    node.visited += 1
26    if node.predecessor_count == 0 || node.visited == node.predecessor_count then
27      foreach( Node nd in node.successors ){ parallelize(nd, knode) ;
28    }
```

---

node and its children in the CFG lie in the same scope. We can then simply add a node prior to and another node right after the barrier-free node. The processing gets involved when a node and its children are in different scopes. In such cases, we need to find the predecessors of these nodes which lie in the same scope.

### 3.3 Data-driven versus Topology-driven Processing

It is a two step process. In step one, each kernel is checked if it can be converted into worklist-based and in step two, AST is modified such that code generation module generates worklist based code. The primary program analysis required in this transformation is to identify where point attributes are getting modified (e.g., distance of nodes in SSSP) and push such points (vertices) into the worklist. Our method goes through the AST of `foreach` and checks for such attribute updates and checks. Based on the target architecture, code generation module generates worklist based code for the particular target. In case of CPU, Falcon generates Galois worklist based code. In case of GPU, Falcon library provides a worklist interface.

### 3.4 CPU, GPU and Multi-GPU Codes

Falcon [2] requires a programmer to write different DSL code for different backends. It uses `<GPU>` tag to specify a GPU variable. Falcon compiler generates

GPU code if there exists a GPU variable in the program and converts function to GPU kernel if one of the parameters is a GPU variable. We modified the Falcon grammar so that compiler does not need a GPU tag. It recognizes a device-independent version of the DSL code. Based on the command-line argument, our compiler generates code for an appropriate target device.

The compiler generates the GPU code in the following manner. First, it marks all the *target functions* as kernels. Second, it marks the global variables used in the *target functions* as GPU variables. Third, it makes a GPU copy of each of the variables of type graph, set and collection. Fourth, it replaces the CPU copy of a graph, set or collection with its corresponding GPU copy.

To generate multi-GPU code, the programmer has to use `parallel sections` construct of Falcon. The Falcon compiler assumes that each of the sections is independent of each other. We identify the number of sections in a parallel sections construct and map each of the sections to a different GPU. For each graph, set and collection used in a particular section, a GPU copy is created in the mapped GPU. It may happen that the programmer has used a single graph and used that graph in multiple sections. In such cases, the graph needs to be copied to each GPU. For each of those GPU copies, we keep track of the attributes of the graph or its points/edges used in the target functions where the graph is passed as an argument. This helps us to replace the graph whose attribute is accessed in CPU by the appropriate GPU copies where the accessing attribute is present. Now if an attribute of a GPU graph is accessed in the CPU, the Falcon compiler generates a call to `cudaMemcpy` to copy the attribute from GPU to CPU or from CPU to GPU based on whether the programmer has read or written to the attribute. One advantage of assuming independent sections is that the attributes accessed in CPU can be changed on maximum one GPU, which eases our analysis and code generation.

## 4 Experimental Evaluation

We modified Falcon’s [2] abstract syntax tree (AST) processing and code generation to generate various types of codes presented in the last section. For CPUs, it generates parallel code with OpenMP for vertex and edge based processing, while Galois worklist code for worklist based processing. For GPU and multi-GPU targets, it generates CUDA code.

### 4.1 Experimental Setup

We used a range of graph types to assess the effectiveness of our proposal. The dataset graphs in our experimental setup and their characteristics are presented in Table 4. We used four graph algorithms in our testbed: Breadth-First Search (BFS), Connected Components (CC), Minimum Spanning Tree computation (MST) and Single-Source Shortest Paths computation (SSSP). All these algorithms are fundamental in graph theory and form building blocks in various application domains. We compare the generated codes against the following

frameworks: Galois [18], Totem [6], Green-Marl [9], LonestarGPU [17] and Gunrock [24]. Our auto-generated codes perform similar to hand-optimized Falcon codes. Therefore, in the sequel, we discuss directly our proposed techniques embedded into existing Falcon, unless otherwise stated.

The CPU benchmarks for OpenMP are run on an Intel XeonE5-2650 v2 machine with 32 cores clocked at 2.6 GHz with 100GB RAM, 32KB of L1 data cache, 256KB of L2 cache and 20MB of L3 cache. The machine runs CentOS 6.5 and 2.6.32-431 kernel, with GCC version 4.4.7 and OpenMP version 4.0. The CUDA code is run on Tesla K40C devices each having 2880 cores clocked at 745 MHz with 12GB of global memory. Eight similar GPU devices are connected to the same CPU device.

Graph	#nodes $\times 10^6$	#edges $\times 10^6$	max- degree
USA-CTR	14	34	9
USA-full	24	58	9
orkut	3	234	33313
sinaweibo	21	261	278491
rand-25M	25	100	17
rand-50M	50	200	18
rand-75M	75	300	18
rand-100M	100	400	18
rand-125M	125	500	19
rmat-10M	10	100	1873
rmat-20M	20	200	2525
rmat-40M	40	400	3333
rmat-50M	50	500	4132

Table 4: Benchmark characteristics

Graph	BFS	CC	MST	SSSP
USA-CTR	3456	14103	727	30299
USA-full	9113	24061	779	72857
orkut	269	623	5509	267
sinaweibo	1234	1955	30556	1151
rand-25M	131	411	2832	561
rand-50M	270	823	6665	1142
rand-75M	414	1416	11265	1796
rand-100M	583	2192	15860	2413
rand-125M	756	2909	25973	3194
rmat-10M	133	387	2770	536
rmat-20M	266	789	6240	1169
rmat-40M	542	1601	13232	2405
rmat-50M	707	2026	16825	3808
<b>Total</b>	<b>17874</b>	<b>53296</b>	<b>139233</b>	<b>121598</b>

Table 5: Baseline times (ms) of Falcon on GPU

## 4.2 Baselines and Comparison with Other Frameworks

The baseline execution times of Falcon on GPU are listed in Table 5. We observe that the execution times on road networks are particularly high for propagation based algorithms such as BFS, SSSP and CC. This occurs because unlike other graphs, road networks have large diameters, leading to many iterations of the algorithm with parallelism not enough for GPU. The opposite occurs for MST.

Figures 1, 2, 3 present the performance benefit of our modified Falcon against other frameworks. The GPU-baseline used for this comparison is Totem, whose speedup is assumed to be 1.0 (hence not shown in the plots). On CPU, the baseline is Galois with one thread. For GPU-SSSP, we observe that Falcon-generated code provides consistently better speedups compared to LonestarGPU and Gunrock, except on the two social networks (orkut and sinaweibo). Totem performs

better on the social networks as well as on RMAT graphs due to its inbuilt edge-based processing and other optimizations to improve load-balancing across GPU threads. For BFS, the results are mixed across various frameworks and there is no clear winner, but there are interesting patterns based on the graph types. Gunrock performs quite well on the road networks (USA-full and USA-CTR), primarily due to its work-efficient worklist-based processing. Totem outperforms again on social networks due to edge-based processing and better load-balancing. Performance of almost all the frameworks on RMAT graphs is quite similar, with LonestarGPU performing poorly. Our Falcon stands out on random graphs with speedups close to  $2\times$  over all other frameworks. On CPU, Galois outperforms other frameworks for SSSP, but Green-Marl is a close second. Note that Galois uses hand-crafted libraries, while Green-Marl is auto-generated, similar to Falcon. Totem performs quite poorly for SSSP, but bounces back for BFS outperforming all the other frameworks. Galois outperforms other frameworks on road networks because it uses a different algorithm (delta-stepping). For CC, Falcon does not perform well on road inputs. Performance of MST on GPU is shown in Figure 5.

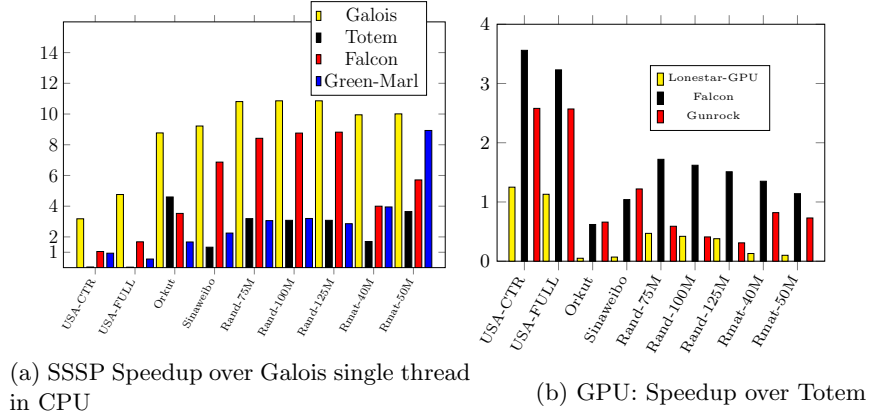
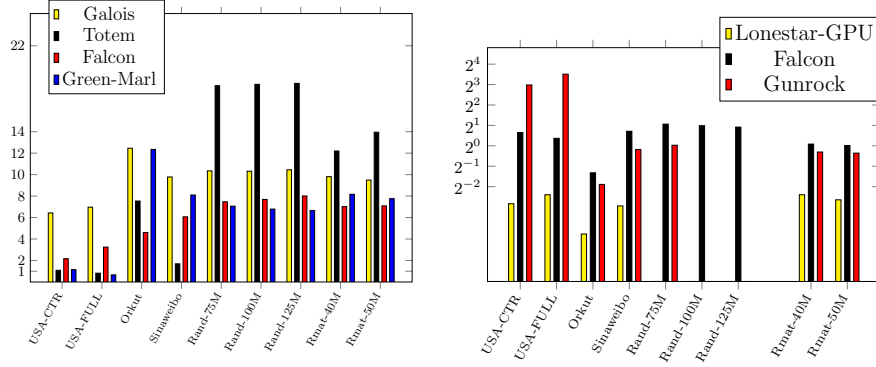


Fig. 1: SSSP comparison

### 4.3 Effect of Vertex-based versus Edge-based

Figures 4a and 4b presents results of edge-based versus vertex-based processing of Falcon across various graphs for CC, BFS and SSSP. We observe that edge-based processing performs better in social-networks (orkut and sinaweibo) and RMAT graphs. Both these kinds of graphs have skewed (power-law) degree-distribution resulting in large load-imbalance with vertex-based processing. These graphs follow small-world property due to this peculiar (and natural) degree distribution. On GPUs, this load-imbalance manifests itself as thread-divergence as the number of iterations (based on the number of neighbors) of each thread has



(a) BFS Speedup Over Galois single thread in CPU (b) BFS Speedup over Totem in GPU

Fig. 2: BFS comparison

high variance. In other words, threads mapped to vertices having few neighbors have to wait for others mapped to high-degree vertices. This inhibits parallelism for SIMT style of processing. In contrast, in edge-based processing, since threads are mapped to (a group of) edges, the load-imbalance is relatively negligible. This results in better thread-divergence among warp-threads, leading to improved execution time. Road networks and random graphs, on the other hand, have quite uniform degree-distribution. Therefore, edge-based processing is not very helpful. In fact, for uniform degree-distributions, edge-based processing may lead to inferior results (as seen in our experiments), due to increased synchronization requirement. Different outgoing edges of a vertex are processed sequentially by the same thread in vertex-based processing; whereas, those are processed in parallel by different threads. Thus, edge-based processing necessitates more coordination among threads with respect to reading and updating attribute values of vertices. The worklist based code does not benefit on GPU. Speedup of  $\Delta$ -Stepping worklist [15] based code is much faster than *OpenMP* library based vertex-based code as shown in figure 4c.

We observe that, unlike on GPUs, edge-based processing is not helpful on CPUs. This is primarily due to CPUs not having enough resources to utilize the additional parallelism exposed by edge-based processing. Thus, a few tens of threads perform in a similar manner in the presence of a million vertices or multi-million edges. The only exception is sinaweibo graph which witnesses over  $3\times$  speedup for CC on CPU due to edge-based processing. The improvements on this graph are also high for other algorithms as well (BFS and SSSP) compared to other graphs. This occurs due to higher average degree in this social network. Higher average degree adds sequentiality in vertex-based processing, while edge-based processing is not amenable to degree-distribution or average degree. The overall effect gets pronounced for such dense graphs.

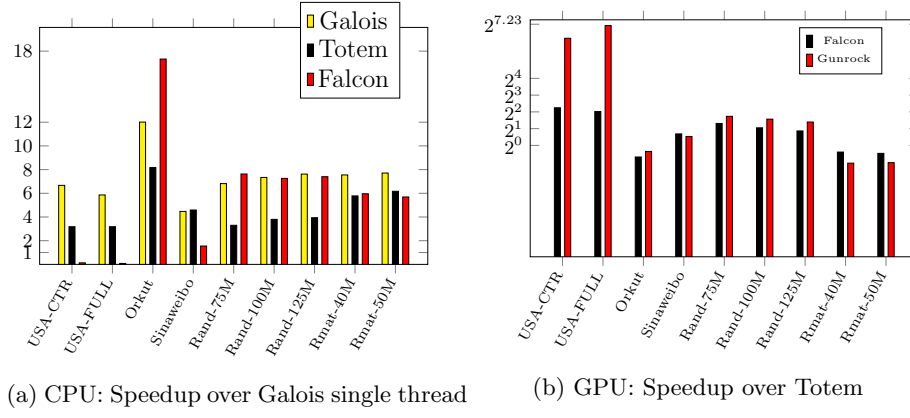


Fig. 3: CC comparison

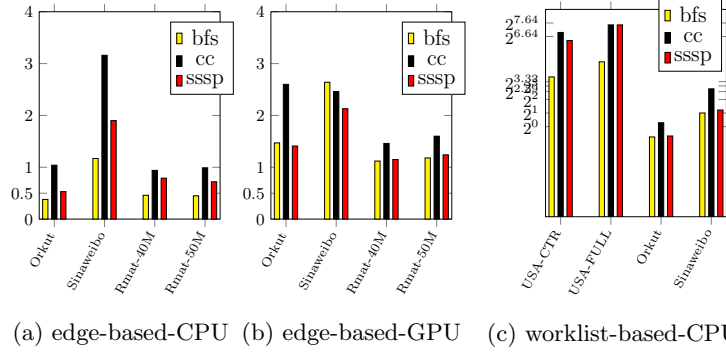


Fig. 4: Speed of edge-based and worklist-based code over vertex-based code

#### 4.4 Effect of Synchronous versus Asynchronous Processing

Graph	Syn- chronous	Asyn- chronous
USA-CTR	86	91
USA-full	134	106
orkut	425	285
sinaweibo	8310	4508
rand-25M	458	347
rand-50M	945	741
rmat-10M	329	320
rmat-20M	771	665
<b>Average</b>	<b>1433</b>	<b>882</b>

Table 6: BFS and SSSP (CPU) Sync. versus Async.

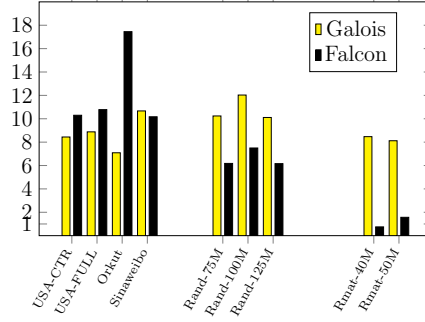


Fig. 5: MST Speedup over Galois Single

Table 6 presents the effect of asynchronous processing for various graphs on CPU. We used a combination of BFS and SSSP to perform independent processing on the same graph. We observe that asynchronous version improves execution time by 38%. This occurs because threads do not have to wait for other threads. This is primarily true on CPUs as threads are monolithically working on different parts of the graph and seldom require synchronization. In our experiments, since all the GPU resources were utilized by a single kernel, asynchronous processing performed similar to synchronous execution.

#### 4.5 Effect of Code Generation for Multiple Targets

Graphs	CPU-16	GPU	multi-GPU
rand-25M + rand50M	3866	1234	826
rmat-10M + rmat20M	3024	1176	792

Table 7: Connected Components

Graphs	CPU-16	GPU	multi-GPU
USA-CTR + USA-full	25363	12569	9138
sinaweibo + orkut	4845	1503	1259

Table 8: Breadth First Search

Fig. 6: Execution time (in ms) of BFS and CC for various targets

Our approach can seamlessly generate code for CPU or GPU or multi-GPU. The multi-GPU code works with different graphs for the same algorithm, or with the same graph for different algorithms. Table 7 presents results for the former with CC as the algorithm generating code for CPU with 16 threads, single GPU and two GPUs, while Table 8 presents those for BFS. We observe that multi-GPU version took much less time as compared to other backends. In both CPU and single-GPU versions, the graphs are processed one after another. On the other hand, in multi-GPU version, both the graphs are processed simultaneously in different GPUs; so the overall execution time is the larger of the two.

## 5 Related Work

Green-Marl [9] is a graph DSL for implementing parallel graph algorithms on multi-core CPUs. Green-Marl was extended for GPUs [20] and CPU clusters [10]. Falcon [2] is a DSL for graph analytics on single machine with one or more GPUs. Falcon is extended for distributed systems with CPU and GPU [1], [22]. Galois [18] is a C++ framework for graph analytics on multi-core CPUs. Ligra [21] is a framework for writing graph traversal algorithms for multi-core shared memory systems.

There are efficient implementations different graph algorithms [14,19,13] on GPU. Worklist-based graph algorithms do not benefit much on GPU [5]. The Lonestar-GPU [17] framework supports dynamic graph algorithms on GPU. The Gunrock [24] framework provides a data-centric abstraction for graph operations with GPU-specific optimizations. Totem [6] is a heterogeneous framework for graph processing for a multi-GPU machine.

Large graphs require processing on computer cluster. GraphLab [11], PowerGraph [7], Pregel [12] and Giraph [3] are popular distributed graph analytics framework. Bulk Synchronous Parallel (BSP) Model [23] of execution and asynchronous executions are popular models of executions. Gluon [4] uses Galois and Ligra and generates distributed-memory versions of these systems.

## 6 Conclusion

Irregular codes have data-dependent access patterns. Therefore, compilers need to make pessimistic assumptions leading to very conservative code. While DSLs for irregular codes allow us the flexibility to make more informed decisions about the domain, existing DSLs lack adaptability. Different graphs expect different kinds of processing to achieve the best performance. While existing DSLs do allow changing the algorithm specification to suit a purpose, it would be ideal if the specification remains intact and the compiler judiciously generates the necessary efficient code. We presented our experiences in achieving the same, for a graph DSL, Falcon. In particular, we auto-generated codes for vertex-based and edge-based, for synchronous versus asynchronous, for worklist-driven versus topology-driven, and for CPU versus GPU versus multi-GPU processing. We illustrated the effectiveness of our techniques using a variety of algorithms and several real-world graphs.

## References

1. Cheramangalath, U., Nasre, R., Srikant, Y.N.: DH-Falcon: A Language for Large-Scale Graph Processing on Distributed Heterogeneous Systems. In: CLUSTER. pp. 439–450. IEEE, USA (Sept 2017), <https://ieeexplore.ieee.org/document/8048957>
2. Cheramangalath, U., Nasre, R., Srikant, Y.N.: Falcon: A graph manipulation language for heterogeneous systems. ACM Trans. Archit. Code Optim. 12(4), 54:1–54:27 (Dec 2015), <http://doi.acm.org/10.1145/2842618>

3. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.* 8(12), 1804–1815 (Aug 2015), <http://dx.doi.org/10.14778/2824032.2824077>
4. Dathathri, R., Gill, G., Hoang, L., Dang, H.V., Brooks, A., Dryden, N., Snir, M., Pingali, K.: Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In: *PLDI*. pp. 752–768. *PLDI 2018*, ACM, New York, NY, USA (2018), <http://doi.acm.org/10.1145/3192366.3192404>
5. Davidson, A., Baxter, S., Garland, M., Owens, J.D.: Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In: *IPDPS*. pp. 349–359. *IPDPS '14*, IEEE Computer Society, Washington, DC, USA (2014), <https://doi.org/10.1109/IPDPS.2014.45>
6. Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., Ripeanu, M.: A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In: *PACT*. pp. 345–354. *PACT '12*, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2370816.2370866>
7. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: *OSDI*. pp. 17–30. *OSDI'12*, USENIX Association, Berkeley, CA, USA (2012), <http://dl.acm.org/citation.cfm?id=2387880.2387883>
8. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: A dsl for easy and efficient graph analysis. In: *ASPLOS*. pp. 349–362. *ASPLOS XVII*, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2150976.2151013>
9. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: A dsl for easy and efficient graph analysis. In: *ASLPOS*. pp. 349–362. *ASPLOS XVII*, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2150976.2151013>
10. Hong, S., Salihoglu, S., Widom, J., Olukotun, K.: Simplifying Scalable Graph Processing with a Domain-Specific Language. In: *CGO*. pp. 208:208–208:218. *CGO '14*, ACM, New York, NY, USA (2014), <http://doi.acm.org/10.1145/2544137.2544162>
11. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5(8), 716–727 (Apr 2012), <http://dx.doi.org/10.14778/2212351.2212354>
12. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. pp. 135–146. *SIGMOD '10*, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1807167.1807184>
13. Mendez-Lojo, M., Burtcher, M., Pingali, K.: A GPU Implementation of Inclusion-based Points-to Analysis. In: *PPoPP*. pp. 107–116. *PPoPP '12*, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2145816.2145831>
14. Merrill, D., Garland, M., Grimshaw, A.: Scalable gpu graph traversal. *SIGPLAN Not.* 47(8), 117–128 (Feb 2012), <http://doi.acm.org/10.1145/2370036.2145832>
15. Meyer, U., Sanders, P.: Delta-stepping: A parallel single source shortest path algorithm. In: *Proceedings of the 6th Annual European Symposium on Algorithms*. pp. 393–404. *ESA '98*, Springer-Verlag (1998), <http://dl.acm.org/citation.cfm?id=647908.740136>
16. Nasre, R., Burtcher, M., Pingali, K.: Data-Driven Versus Topology-driven Irregular Computations on GPUs. In: *Proceedings of the 2013 IEEE 27th*

- International Symposium on Parallel and Distributed Processing. pp. 463–474. IPDPS '13, IEEE Computer Society, Washington, DC, USA (2013), <https://doi.org/10.1109/IPDPS.2013.28>
17. Nasre, R., Burtscher, M., Pingali, K.: Morph Algorithms on GPUs. *SIGPLAN Not.* 48(8), 147–156 (Feb 2013), <http://doi.acm.org/10.1145/2517327.2442531>
  18. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X.: The Tao of Parallelism in Algorithms. *SIGPLAN Not.* 46(6), 12–25 (Jun 2011), <http://doi.acm.org/10.1145/1993316.1993501>
  19. Sariyüce, A.E., Kaya, K., Saule, E., Çatalyürek, U.V.: Betweenness Centrality on GPUs and Heterogeneous Architectures. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. pp. 76–85. GPGPU-6, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2458523.2458531>
  20. Shashidhar, G., Nasre, R.: LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs. In: *Languages and Compilers for Parallel Computing (LCPC)*. pp. 1–10. IEEE, USA (2016)
  21. Shun, J., Blelloch, G.E.: Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.* 48(8), 135–146 (Feb 2013), <http://doi.acm.org/10.1145/2517327.2442530>
  22. Upadhyay, N., Patel, P., Cheramangalath, U., Srikant, Y.N.: Large scale graph processing in a distributed environment. In: *Euro-Par 2017: Parallel Processing Workshops*. pp. 465–477. Springer International Publishing, Cham (2018)
  23. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* 33(8), 103–111 (Aug 1990), <http://doi.acm.org/10.1145/79173.79181>
  24. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.* 51(8), 11:1–11:12 (Feb 2016), <http://doi.acm.org/10.1145/3016078.2851145>