Graph algorithms have been shown to possess enough parallelism to keep several computing resources busy—even hundreds of cores on a GPU. Unfortunately, tuning their implementation for efficient execution on a particular hardware configuration of heterogeneous systems consisting of multi-core CPUs and GPUs is challenging, time-consuming, and error-prone. To address these issues, we propose a Domain Specific Language (DSL), Falcon, for implementing graph algorithms that (i) abstracts the hardware, (ii) provides constructs to write explicitly parallel programs at a higher level, and (iii) can work with general algorithms that may change the graph structure (morph algorithms). We illustrate the usage of our DSL to implement local computation algorithms (that do not change the graph structure) and morph algorithms such as Delaunay mesh refinement, survey propagation and dynamic SSSP on GPU and multi-core CPU. Using a set of benchmark graphs, we illustrate that the generated code performs close to the state-of-the-art hand-tuned implementations.

1. INTRODUCTION

Graphs model relationships across real-world entities in web graphs, social network graphs, and road network graphs. Graph algorithms analyze and transform a graph to discover graph properties or to apply a computation. For instance, a pagerank algorithm computes a rank for each page in the webgraph, a community detection algorithm discovers likely communities in a social network, while a shortest path algorithm computes the quickest way to reach from one place to another in a road network.

An algorithm is irregular if its data-access pattern or control-flow pattern is unpredictable at compile time. Static analysis techniques prove inadequate to deal with the analysis and parallelization of irregular algorithms, and we require dynamic techniques [Pingali et al. 2011] to deal with such situations. Traditionally, graph algorithms have been perceived to be difficult to analyze as well as parallelize because they are irregular.

GPUs further complicate graph algorithm implementations: managing separate memory spaces of CPU and GPU, SIMD (single instruction multiple data) execution, exposed thread hierarchy, asynchronous CPU/GPU execution, etc. Hand-written and efficient implementations are not only difficult to code and debug, but are also error prone.

It would be really helpful if a programmer can specify a graph algorithm in a hardware-independent manner and focus solely on the algorithmic logic. Unfortunately, such an approach—which essentially auto-parallelizes a sequential piece of code—provides limited performance in general when compared with a manually parallelized hardware-centric code by an expert.
Our goal in this work is to bridge this performance gap between an auto-generated code and a manually crafted implementation. We wish to let the programmer write the algorithm at a higher level (much higher than CUDA and OpenCL), without any hardware-centric constructs. To achieve performance close to that of a hand crafted code, we make two compromises: (i) we allow only graph algorithms to be specified (i.e., we do not provide special constructs for other type of algorithms), and (ii) we require the code to be explicitly parallel. The first compromise trades generality for speed, while the second one allows our code generator to emit hardware-specific code.

Our specific contributions are given below.

— The design of Falcon, a DSL for general graph algorithms. Unlike previously reported languages, Falcon supports morph algorithms, that is, algorithms wherein the graph structure may also change, apart from the values at the nodes and the edges.

— Falcon's code generation scheme for multicore CPU, single GPU, multi-GPU, and heterogeneous backends. Our compiler supports worklist based implementations of morph and local computation algorithms on CPU, that run much faster than most hand-written implementations.

— Falcon's support for graph partitioning and execution of a single algorithm on the partitioned graph on the CPU and multiple-GPUs (for vertex-centric algorithms only).

— Performance analysis of Falcon: We generate CUDA and OpenMP code for morph algorithms such as Delaunay mesh refinement (DMR), survey propagation (SP) and dynamic single source shortest path (SSSP), as well as CUDA and OpenMP code for local computation algorithms. Performance of these and several other benchmarks are compared against the state-of-the-art DSL and framework-based implementations.

Rest of the paper is organized as follows. Section 2 mentions the benefits of Falcon. Section 3 compares and contrasts with related work. We present the Falcon language with example programs in Section 4. Section 5 explains the code generation phase of the compiler. Section 6 discusses the performance evaluation of the code generated by the Falcon compiler, and we conclude in Section 7.

2. BENEFITS OF FALCON

Existing DSLs such as Green Marl [Hong et al. 2012] and Elixir [Prountzos et al. 2012] auto-parallelize sequential graph algorithm implementations. The algorithm specification in these DSLs tends to be much smaller and simpler compared to the corresponding specification in a general purpose language such as C or Python. However, there are multiple issues with the existing approaches. First, they target only a single type of device (multicore CPUs). It is unclear if these frameworks can be modified to effectively support heterogeneous systems. Second, their scope is limited to graph analytic algorithms, wherein the graph structure is assumed to be static. Therefore, the domain of morph algorithms is unsupported. As has been shown earlier [Nasre et al. 2013b], concurrent execution of morph algorithms poses new challenges, and their efficient parallel execution is quite difficult. Third, despite the simplicity of these DSLs, a user needs to invest time in learning a new language. This last issue is addressed by library based approaches such as Galois [Pingali et al. 2011] and Totem [Gharaibeh et al. 2013]. However, Totem does not support morph algorithms, while Galois does not work for heterogeneous systems. New challenges while dealing with GPUs and heterogeneous systems in the context of auto-parallelization of structural graph updates are not addressed in any existing framework.

Falcon supports both morph and local computation algorithms for GPU, multi-GPU and a combination of CPU and GPU. It extends the C language, and provides a rich set of constructs and concurrent data structures for efficient execution across computing systems. Unlike Green Marl and Elixir, Falcon also allows a user to write the entry function main allowing him full control over the program’s execution. In Falcon, it is easy to write a worklist based implementation of many algorithms on the multicore-CPU which are much faster than the state-of-the-art implementations (for example, the Δ-stepping SSSP algorithm [Meyer and Sanders 1998] implementation).
Writing code for GPU based algorithms is very simple in Falcon. A programmer is simply required to annotate the location of the graph object, using an optional <GPU> tag, and the rest, including thread, device, and memory management is handled by the Falcon compiler. The parallel sections in Falcon can be used to specify concurrent execution of CUDA kernels on different GPU devices. Generation of code for CPU is equally easy in Falcon. Further, the support for execution of vertex-centric algorithms on partitioned graphs makes such implementations easy for very large graphs that do not fit entirely in GPU memory.

Handwritten codes of LonestarGPU [Nasre et al. 2013b] for GPU and Galois [Pingali et al. 2011] for multicore CPU, both of which support morph algorithms, are very complex. Coding a new algorithm using these platforms requires a very good knowledge of the device architecture, thread management and memory management and the programmer is required to handle all these on his/her own. Such a code is difficult to debug. This makes Falcon as a new choice for coding parallel graph algorithms that is easy to use, debug, and is also efficient.

3. RELATED WORK

<table>
<thead>
<tr>
<th>References</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green-Marl [Hong et al. 2012], Elixir [Prountzos et al. 2012], [Hong et al. 2014]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[Ragan-Kelley et al. 2013]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Lonestar-GPU [Nasre et al. 2013b]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>[Shun and Blelloch 2013], [Roy et al. 2013], [Zhang et al. 2015]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Medusa [Zhong and He 2014], [Lee et al. 2009]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Totem [Gharaibeh et al. 2013][Gharaibeh et al. 2012]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Galois[Pingali et al. 2011]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>[Feng et al. 2012], [Menon et al. 2012]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[Tian et al. 2011] [Tian et al. 2008]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>[Low et al. 2012], [Bader and Madduri 2008], [Gregor and Lumsdaine 2005]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

Table I. Related Works Comparison
A=DSL, B=Framework, C=Library, D=CPU, E=GPU, F=Speculation

Green-Marl [Hong et al. 2012] and Elixir [Prountzos et al. 2012] are examples of Graph DSLs, and both of them target multicore CPU. Green-Marl and Elixir can be used to implement only local computation algorithms.

Morph algorithms can be classified as cautious, if the algorithms read all the neighborhood elements before modifying any of them. The Galois framework [Pingali et al. 2011], which is a library implementation in C++, supports cautious morph algorithms and generates code only for multicore CPU. Cautious morph algorithms have been implemented on GPU by Nasre et al.
GraphLab [Low et al. 2012] is a framework that supports a combination of machine learning and graph algorithms. Pregel [Malewicz et al. 2010] is a graph-processing framework in a distributed setting. It uses bulk-synchronous parallelism for efficient execution of graph algorithms in a cluster of nodes. OpenMP to GPGPU [Lee et al. 2009] is a framework for automatic code generation for GPU from OpenMP CPU code. The Medusa [Zhong and He 2014] framework generates CUDA code using device APIs for graph elements and supports multi-GPU systems. Paragon [Samadi et al. 2012] uses GPU for speculative execution and on misspeculation, that part of the code is executed on CPU. An online profiling based method by [Kaleem et al. 2014] partitions work and distributes it across CPU and GPU.

Parallel Boost Graph Library [Gregor and Lumsdaine 2005] is a distributed version of BGL and SNAP [Bader and Madduri 2008] [Bader and Madduri 2005] is a stand-alone parallel graph analysis package. CuSha [Khorasani et al. 2014] proposes two new ways of storing graphs on GPU that has improved regular memory access patterns. Efficient implementations of local computation algorithms such as Breadth First Search (BFS) and Single Source Shortest Path (SSSP) on GPU have been reported several years ago [Harish and Narayanan 2007] [Harish et al. 2009]. There have also been successful implementations of other local computation algorithms such as n-body simulation [Burtscher and Pingali 2011], betweenness centrality [Sariyüce et al. 2013] and data flow analysis [Mendez-Lojo et al. 2012; Prabhu et al. 2011] on GPU. [Davidson et al. 2014] proposes different ways of writing SSSP programs on GPU along with their merits and demerits. It concludes that worklist-based implementation would not benefit much on GPU compared to that on a CPU.

iGPU [Menon et al. 2012] architecture proposes a method for breaking a GPU function execution into many idempotent regions so that in between two continuous regions, there is very little live state and this fact can be used for speculative execution. Min Feng et al. [Feng et al. 2012] implemented methods for speculative parallelization of loops on GPU which have irregular memory access and control flow. The CoRD [Tian et al. 2008, 2011] framework proposes methods for speculative execution on multicore CPU. It supports rollbacks and morph algorithms which need not be cautious. More references related to Graphs, Graph DSLs, Speculation etc., can be found in Table I. Falcon currently supports only cautious morph algorithms.

4. OVERVIEW OF FALCON

4.1. Introduction

Falcon is a graph DSL and it extends the C programming language. In addition to the full generality of C (including pointers, structs and scope rules), Falcon provides the following types relevant to graph algorithms: Point, Edge, Graph, Set and Collection. It also supports constructs such as foreach and parallel sections for parallel execution, single for synchronization, and reduction operations. Many complete examples of Falcon programs are available in [Unnikrishnan et al. 2015].

4.2. Example: Shortest Path Computation

Single source shortest path (SSSP) computation is a fundamental operation in graph algorithms. Given a designated source node, an SSSP algorithm computes the shortest distance from the source node to each node. Figure 1 shows the code for SSSP computation in Falcon for GPU. The algorithm first initializes the dist variable of all the nodes to a large value (Line 24). The dist variable of the source node is then made zero (Line 25). It then progressively relaxes nodes to determine whether there is any shorter path to a node via some other incoming edge (Line 29). This is done by checking the condition (for each edge \((u, v)\) \(\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)\)). If this condition is satisfied, then the distance of the destination node \(v\) is changed to the smaller value via \(u\) (Line 5), using an atomic operation (more on this later). This procedure is repeated until we reach a fix point (lines 27-32).
Falcon needs each variable which resides on the GPU to have the <GPU> tag preceding the variable name in the declaration statement (Lines 1, 19). Being a graph-DSL, the type Graph is directly available in the language.

```c
int <GPU> changed = 0; // Variable on GPU

relaxgraph(Point <GPU> p, Graph <GPU> graph) {
   p.uptd = false;
   foreach (t In p.outnbrs) {
      MIN(t.dist, p.dist + graph.getWeight(p, t), changed);
   }
}

reset(Point <GPU> t, Graph <GPU> graph) {
   t.dist = t.olddist = 1234567890; t.uptd = false;
}

reset1(Point <GPU> t, Graph <GPU> graph) {
   if (t.dist < t.olddist) t.uptd = true;
   t.olddist = t.dist;
}

//main function on rhs
main(int argc, char *argv[]) {
   Graph hgraph; // graph on CPU
   hgraph.addPointProperty(dist, int);
   hgraph.getType() <GPU> graph; // graph on GPU
   graph.addPointProperty(uptd, bool);
   graph.addPointProperty(olddist, int);
   hgraph.read(argv[1]) // read graph on CPU
   graph = hgraph; // copy graph to GPU
   foreach (t In graph.points) reset(t, graph);
   foreach (t In graph.points) reset1(t, graph);
   while (1) {
      changed = 0; //keep relaxing on GPU
      foreach (t In graph.points) (t.uptd)
         relaxgraph(t, graph);
      if (changed == 0) break; //reached fix point
      foreach (t In graph.points)
         reset(t, graph);
      hgraph.dist = graph.dist; // copy all points
dist to CPU
      for (int i = 0; i < hgraph.npoints; ++i)
         printf("i=%d dist=%d
", i,
         hgraph.points[i].dist);
   }
}
```

**Fig. 1:** Optimized GPU SSSP code in Falcon

Line 18 adds a property dist to each Point in the CPU Graph object, hgraph. The getType() function on Line 19 (a compile-time function) returns a type which is used to create a Graph object graph on the GPU. An object created from another type also inherits its dynamic properties. Thus, the object graph automatically gets dist property attached to its points. Lines 20–21 add two properties (uptd, olddist) to points in the GPU Graph object graph. Lines 22–23 read the graph from a file into CPU memory and copy it to the GPU memory. The compiler generates efficient code to perform this copy operation using DMA.

GPU kernels are specified using a foreach construct. Line 24 uses the foreach parallelizing construct to initialize a few properties of each Point in the graph variable. The foreach statement identifies that the Graph object it uses is on the GPU and the appropriate GPU code is generated automatically. The compiler needs to (i) identify the kernel code, (ii) identify the variables used in the computation, and (iii) pass the appropriate parameters.

The relaxgraph() function is called repeatedly (Line 29) and it keeps on reducing dist value of each Point (Line 5). The foreach in relaxgraph() is augmented with a condition (t.uptd) that makes sure that only those points which satisfy the condition will execute the code inside the relaxgraph() function. In the first invocation of relaxgraph(), only the source node will perform the computation. Since multiple threads may update the distance of the same node (e.g., when relaxing edges \((u_1, v)\) and \((u_2, v)\)), some synchronization is required across the threads. This is achieved by providing atomic variants for commonly used operations. The MIN() function used by relaxgraph()
Table II. Data Types in Falcon

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Major Fields</th>
<th>Major Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>Point in Graph</td>
<td>x, y, z</td>
<td>del(), getNeighbours()</td>
</tr>
<tr>
<td>Edge</td>
<td>Edge in Graph</td>
<td>src, dst, weight</td>
<td>del()</td>
</tr>
<tr>
<td>Graph</td>
<td>Entire Graph</td>
<td>points[], edges[], npoints, nedges</td>
<td>addEdge(), addPoint(), getWeight(), read(), addEdgeProperty(), sortEdges(), addProperty(), makePartition(), updatePartition()</td>
</tr>
<tr>
<td>Set</td>
<td>A static collection</td>
<td>size, parent</td>
<td>find(), union(), clear()</td>
</tr>
<tr>
<td>Collection</td>
<td>A dynamic collection</td>
<td>size</td>
<td>add(), del(), orderByIntValue(), clear()</td>
</tr>
</tbody>
</table>

is an atomic function that reduces \( dist \) atomically (if necessary) and if it does change, the third argument value will be set to 1 (Line 5). So, whenever there is a reduction in the value of \( dist \) for even one Point, the variable \( \text{changed} \) is set to 1. Line 3 makes \( \text{uptd} \) property of each Point whose current value is \( \text{true} \) to \( \text{false} \). After each call to \( \text{relaxgraph()} \), the \( \text{reset1()} \) function makes \( \text{uptd} \) true only for points whose distance from the source node was reduced in the last invocation of the \( \text{relaxgraph()} \) function (Line 31). The variable \( \text{changed} \) is reset to zero before \( \text{relaxgraph()} \) is called in each iteration (Line 28). Its value is checked after the call and if it is zero, indicating a fixed-point, the control leaves the while loop (Line 30). At this stage, the computation is over. The final \( dist \) value of each Point is copied from the GPU to the CPU in Line 33 (this is also a DMA transfer). The final \( dist \) value of each Point is printed using a for loop in Line 34.

The CPU version of SSSP in Falcon does not differ much from the code in Figure 1. The \(<\text{GPU}>\) tag does not precede any variable name, and there will be only one Graph object. So the code up to Line 18 is the same, with the exception that there is no \(<\text{GPU}>\) tag. The Lines 20 and 21 should be modified to add the properties to CPU graph object \( \text{hgraph} \). There is no need to create a GPU graph object and we should replace all occurrences of the GPU graph object \( \text{graph} \) with the CPU graph object \( \text{hgraph} \). Lines 19, 23 and 33 will be absent in CPU SSSP code.

This example shows the ease of programming in Falcon. A programmer need not worry about memory allocation and thread management on the device. Data copy between the CPU and the GPU is performed efficiently and automatically for basic data types.

4.3. Data Types in Falcon

Table II shows a list of special data types in Falcon along with their important fields and functions.

4.3.1. Point and Edge. A Point data type can have up to three dimensions. An Edge can be directed or undirected and both Point and Edge can store either integer or floating point values in their fields. The Falcon compiler decides all these choices based on command line arguments (input and other options) and does not allocate separate fields for each choice. Functions for Point and Edge are self explanatory.

4.3.2. Graph. A Graph stores its points and edges in vectors points[] and edges[]. The method \( \text{addEdgeProperty()} \) is used to add a property to each edge in a Graph object with the same syntax as \( \text{addPointProperty()} \) used in Line 18 of Figure 1. The \( \text{addProperty()} \) method is used to add a new property to a Graph object (not to each Point or Edge). This will become a property of the whole Graph object. Such a facility allows a programmer to maintain additional data structures with the graph which are not necessarily direct functions of points and edges. For instance, such a function is used in Delaunay Mesh Refinement (DMR) [Chew 1993] code as the graph consists of a collection of \( \text{triangles} \), each \( \text{triangle} \) with three Points and a few
extra properties. The statement shown below illustrates the way DMR code uses this function for a 
Graph object, hgraph.

\[ hgraph.addProperty(triangle, struct node); \]

The structure node has all the fields which are needed for the triangle property of the Graph 
object. This will add to hgraph, a new iterator triangle and a field ntriangle which stores the number 
of triangles.

4.3.3. Set. A Set is an aggregate of unique elements (e.g., a set of threads, a set of nodes, etc.). A 
Set has a maximum size and it can not grow beyond that size. Such a set is naturally implemented 
as a union-find data structure and we have also implemented it as suggested in [Stockel and Bog 
2008], with our own optimizations. The parent field of a Set stores the representative key of 
each element in a Set. A Set data type can be used to implement, as an example, Boruvka’s MST 
algorithm [Chung and Condon 1996]. The way Set data type is declared in MST code is shown in 
Figure 3.

Line 2 declares objects of Set data type one each on CPU and GPU. Each Set object contains a 
set of all the points in the host (hset) and the device (set) Graph objects hgraph and graph 
respectively. As edges get added to the MST, the two end points of the Edge are union-ed into a single 
Set. The algorithm terminates when the Set has a single representative (assuming that the graph 
is connected) or when no edges get added to the MST in an iteration (for a disconnected graph). We 
mark all the edges added to the MST by using the Edge property mark of the Graph object. This 
makes the algorithm a local computation, as the structure of the Graph does not change.

```c
1 minset (Point <GPU> P, Graph <GPU>graph, Set set[Point(graph)]) {
    //finds an Edge with minimum weight from the Set to which Point P belongs to 
a different Set
2 }
3 mstunion (Point <GPU> P, Graph <GPU>graph, Set set[Point(graph)]) {
    //union the Set of Point P with the Set of 
    //Point P’ such that Set(P) != Set(P’) and 
    //Edge(P, P’) is the minimum weight 
    //edge of P, going to different Set 
    //performed only for the Point P that 
satisifes this condition.
4 }
//MinEdge on rhs

MinEdge (Point <GPU> p, Graph <GPU> graph, Set set[Point(graph)]) {
    Point(graph) t1, (graph)t2;
    int t3;
    Edge(graph) e;
    t1 = set.find(p);
    foreach (t In p.outnbrs) {
        t2 = set.find(t);
        t3 = graph.getWeight(p, t);
        if (t1 != t2) {
            if (t3 == t1.minppty.weight) {
                single (t1.minppty.lock) {
                    e = graph.getEdge(p, t);
                    e.mark = true;
                }
            }
        }
    }
}
```

Fig. 2: Finding the minimum weight edge in MST computation

Figure 2 shows how minimum weight edges are marked in the MST computation. Function 
MinEdge(), which gets converted to a device function, takes three parameters: a Point to operate 
on, the underlying Graph object on the GPU, and a Set of points. Line 10 takes each outgoing 
neighbor of the Point p and checks whether those neighbors and p belong to the same set using
the find() function. If not (Line 14), the code checks whether the edge \((p, t)\) has the minimum weight (Line 15). If it is indeed of minimum weight, the code tries to lock the Point using the single construct (see Section 4.5) in Line 16. If the locking is successful, this edge is added to the MST. After MinEdge() completes, each end-point of the edge which was newly added to the MST is put into the same Set using the union operation (performed in the caller).

4.3.4. Collection. A Collection refers to a multiset. Thus, it allows duplicate elements to be added to it and its size can vary (no maximum limit like Set). The extent of a collection object defines its implementation. If its scope is confined to a single function, then we use an implementation based on dynamic arrays. On the other hand, if a collection spans multiple function / kernel invocations, then we rely on the implementation provided by the Thrust library [Nathan Bell (NVIDIA) 2011] for GPU and Galois worklist and its run-time for multicore-CPU. Usage of Galois worklist for multicore-CPU made it possible to write many efficient worklist based algorithms in Falcon. Implementation of operations on Collection such as reduction and union will be done in the near future.

Delaunay Mesh Refinement [Chew 1993] needs local Collection objects to store a cavity of bad triangles and to store newly added triangles. A Collection can be declared in the same way as a Set. A programmer can use add() and del() functions to operate on it and the current length of a Collection can be found using the size field of the data type.

4.4. Variable Declaration

Variable declarations in Falcon can occur in two forms as shown with Point variables \(P_0\) and \(P_1\) below (Edge declarations are similar). Given a Graph object \(g\), we say that \(g\) is the parent of the points and edges in \(g\).

Point \(P_1\), (graph)\(P_0\); //parent Graph of \(P_0\) is \(graph\)

When a point or edge variable has a parent Graph object, it can be assigned values from that parent only and whatever modifications we make to that object will be reflected in the parent Graph object. In the above example, \(P_0\) can be assigned values that are Point objects of \(graph\) only (see also line 6 of Figure 2). But if a variable is declared without a parent and a value is assigned to it, it will be copied to a new location and any modification made to that object will not be reflected anywhere else (e.g., \(P_1\) in the above example).

Falcon allows a programmer to specify on which GPU device the variable needs to be allocated with the optional integer argument along with <GPU> tag. Falcon has a new keyword named struct_rec, that is used to declare recursive data structures. In C, a recursive data structure can be implemented using pointers and the malloc() library function. With struct_rec, a programmer can support a recursive data structure without explicitly using pointers, (like in Java).

4.5. Parallelization and Synchronization Constructs

Falcon provides reduction operations and three statements, single, foreach, and parallel sections to exploit the parallelism available in the GPU.

4.5.1. single statement. This statement is used for synchronization across threads. It ensures mutual exclusion for the participating threads. In graph algorithms, we use single statement to lock a set of graph elements, as discussed later in this section.

When compared to other synchronization constructs such as synchronized construct of Java or lock primitives in pthreads library the single construct differs in two aspects: (i) it has a non-blocking entry, and (ii) only one thread executes the code following it.

Falcon supports two variants for single, as given in Table III: with one item and with a Collection of items. In both the variants the else block is optional (Figure 2, Line 16). The first
variant tries locking one item. As it is a non-blocking entry function, if multiple threads try to get a
lock on the same object, only one will be successful, others will fail. In the second variant, a thread
tries to get a lock on a Collection of items given as an argument. This allows a programmer
to implement cautious forms of algorithms wherein all the shared data (e.g., a set of neighboring
nodes) are locked before proceeding with the computation. A thread succeeds if all the elements
in the Collection object are locked by that thread. As an example, a thread in DMR code tries
to get a lock on a cavity, which is a Collection of triangles. In both the variants, the thread
that succeeds in acquiring a lock executes the code following it and if the optional else block is
present, all the threads that do not acquire the lock execute the code inside the else block.

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

Table III. Single statement in Falcon

4.5.2. foreach statement. This statement is one of the parallelizing constructs in Falcon. It
processes a set of elements in parallel. This statement has two variants as shown in Table IV.
The condition and advance_expression are optional for both the variants. Use of a
condition was explained in Figure 1. An advance_expression is used to iterate from a
given position instead of from the starting or ending positions. A + advance_expression
(- advance_expression, respectively) makes the iterations go in the forward (backward, re-
spectively) direction, starting from the position given by the value of advance_expression.
The advance_expression is optional and its default value is taken as 0. The object used by
foreach (see Table IV) can also be a dereference of a pointer to an object. For examples on use of
these two features of Falcon, the reader is referred to the CPU code of Boruvka MST and DMR in
[Unnikrishnan et al. 2015]. Iterators used in foreach statement for different Falcon data types
are shown in Table V.

<table>
<thead>
<tr>
<th>foreach(item (advance_expression) In object) iterator</th>
<th>Used for Point, Edge and Graph objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach(item (advance_expression) In object) [block of code]</td>
<td>Used for Collection and Set object</td>
</tr>
</tbody>
</table>

Table IV. foreach statement in Falcon

4.5.3. parallel sections. The parallel sections block statement consists of one or
more sections. Each section inside parallel sections runs as a separate parallel re-
gion. With this facility, Falcon can support multi-GPU systems and concurrent execution of CUDA
kernels and parallel execution of CPU and GPU code is possible. Falcon DSL code used to compute
### Table V. Iterators for foreach statement in Falcon

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Iterator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>points</td>
<td>iterate over all points in graph</td>
</tr>
<tr>
<td>Graph</td>
<td>edges</td>
<td>iterate over all edges in graph</td>
</tr>
<tr>
<td>Graph</td>
<td>pptyname</td>
<td>iterate over all elements in new ppty.</td>
</tr>
<tr>
<td>Point</td>
<td>nbrs</td>
<td>iterate over all neighboring points</td>
</tr>
<tr>
<td>Point</td>
<td>innbrs</td>
<td>iterate over src point of incoming edges (Directed Graph)</td>
</tr>
<tr>
<td>Point</td>
<td>outnbrs</td>
<td>iterate over dst point of outgoing edges (Directed Graph)</td>
</tr>
<tr>
<td>Edge</td>
<td>nbrs</td>
<td>iterate over neighbor edges</td>
</tr>
<tr>
<td>Edge</td>
<td>nbr1</td>
<td>iterate over neighbor edges of Point P1 in Edge(P1,P2) (Directed Graph)</td>
</tr>
<tr>
<td>Edge</td>
<td>nbr2</td>
<td>iterate over neighbor edges of Point P2 in Edge(P1,P2) (Directed Graph)</td>
</tr>
</tbody>
</table>

BFS and SSSP distance values for one input graph using parallel sections and multiple GPU Graph objects can be found in Section 5.5.

#### 4.5.4. Reduction Operations
Reduction operations such as ReduxSum, which sums a set of items and ReduxMul which multiplies a set of items are provided by Falcon. We leave the support for arbitrary associative functions as reduction operations as future work.

#### 4.6. Library Functions
We provide atomic library functions MIN, MAX, SUB, AND, etc., which are abstraction over the similar one in CUDA [Nickolls et al. 2008] and GCC [Stallman et al. 2011]. MIN atomic function was used in Figure 1. We also provide a barrier() function which acts as a barrier for the entire group of threads in a CUDA kernel and OpenMP parallel region. A genericbarrier() which supports barriers for a group of related threads is also available.

#### 4.7. Graph Partitioning
Falcon provides support for graph partitioning and execution of vertex-centric algorithms on the CPU and multiple GPUs. This involves partitioning the input Graph into two or more subgraphs and allocation of each subgraph on a GPU or a CPU. This is needed for input graphs that do not fit in the global memory of a single GPU. An algorithm may benefit by executing on both highly multithreaded GPUs and the CPU with the help of a graph partitioning algorithm using the Bulk Synchronous Parallel (BSP) model of execution [Valiant 1990].

#### 5. CODE GENERATION
We now explain how the Falcon compiler generates code (code fragments are shown with macro statements to make the code readable, but these macros are not a part of the compiler generated code). Falcon extends the C language grammar to support additional constructs. The compiler generates CUDA/C++ code. Currently, it supports two types of graph representation: (i) Compressed Sparse Row (CSR) format, and (ii) Coordinate List (COO) or Edge List format. Graphs are stored as C++ classes in Falcon generated code. The GGraph and HGraph classes are used to store a graph on the GPU and the CPU respectively, and both inherit from a parent Graph class. The Graph class has an extra field (of type void *) which stores all the properties added to a Graph object using addPointProperty(), addEdgeProperty(), and addProperty(). The Point and Edge data types can have either integer (default) or floating point values and are stored in a union type with fields ip and fp respectively. The generated code is compiled with nvcc and g++. Figure 4 gives an overview of how parallelization and synchronization is done for CPU and GPU. The Falcon compiler names for all data types and functions specific to CPU and GPU start with H(Host) and G(Gpu) respectively in the generated code.
5.1. Type Checking

Falcon is strongly typed. The compiler checks for undeclared variables, type mismatch involved in an assignment, invalid iterator usage, invalid field-access, invalid property, and usage of the supported data-types (such as Collection).

5.2. Properties

Point and Edge are converted to integer ids. All the extra properties of a Graph object are stored in the extra field, and can be type casted to any structure. By default, extra properties are stored in a structure with the name struct_objectname and are assigned to the extra field of a Graph object. If a Graph object is created by getType() function, its extra properties are assigned to a structure with the name struct_parentobjectname, which will have fields for extra properties of the parent object and all the objects created by the getType() compile time function. In the SSSP example (Figure 1), Graphs on GPU and CPU are both allocated in a structure with the same name as the GPU Graph object is being created with a call of getType(). Figure 5 shows how extra properties of the Graph object on the GPU in the SSSP computation are allocated. For the CPU Graph object (hgraph), only dist field is allocated using malloc(), as olddist and updt fields are associated only with the GPU Graph object(graph). Such simple optimizations are performed during the storage allocation phase of the Falcon compiler.

Fig. 5: Allocating extra property for Graph object on GPU
5.3. Set and Collection

The Falcon compiler has two C++ classes HSet and GSet which implement the CPU and GPU Set data types (resp.). Each of these classes has the same functions named, union to union two sets and find to find the representative key for an element. By default, the key for a subset will be an integer number, which denotes the maximum value of an element in that subset.

Collections that are confined to a kernel are implemented using dynamic arrays. A Collection that spans across multiple functions is implemented using the Thrust library (for GPU), and the Galois worklist along with its runtime code (for CPU). This made possible the worklist based implementation of boruvka MST and SSSP algorithms in Falcon DSL very easy. Details of a Δ-stepping based implementation of the SSSP algorithm in Falcon and the code generated by the Falcon compiler Using the Galois worklist can be found [Unnikrishnan et al. 2015]. A Collection based BFS implementation on GPU (written in Falcon) can be found in [Unnikrishnan et al. 2015].

5.4. Foreach Statement

Code generation for a foreach statement depends on the object on which it is called and where (GPU/CPU) the object is allocated. Nested parallelism using foreach is not supported. We convert inner foreach statements of nested foreach statements to simple for loop statements during code generation.

The outermost loop is retained as a foreach statement and is converted to a CUDA kernel call / OpenMP pragma (except for Collection on CPU) in the generated code. Figure 6 shows the code generated for the relaxgraph() function and its foreach call from Figure 1, with the target being GPU. Since foreach statement inside relaxgraph() is nested inside foreach statement from main(), the foreach inside relaxgraph() is converted to a simple for loop. The variable TPB (Threads Per Block) corresponds to (MaxThreadsPerBlock - MaxThreadsPerBlock % CoresPerSM) for the GPU device on which the CUDA kernel is being called. We also make sure that a kernel executes by splitting a kernel call into multiple calls, if the number of threads or blocks for the kernel call is above the allowed value for device. Each Edge in Falcon stores two values in the edges array, the

---

5.3. Set and Collection

The Falcon compiler has two C++ classes HSet and GSet which implement the CPU and GPU Set data types (resp.). Each of these classes has the same functions named, union to union two sets and find to find the representative key for an element. By default, the key for a subset will be an integer number, which denotes the maximum value of an element in that subset.

Collections that are confined to a kernel are implemented using dynamic arrays. A Collection that spans across multiple functions is implemented using the Thrust library (for GPU), and the Galois worklist along with its runtime code (for CPU). This made possible the worklist based implementation of boruvka MST and SSSP algorithms in Falcon DSL very easy. Details of a Δ-stepping based implementation of the SSSP algorithm in Falcon and the code generated by the Falcon compiler Using the Galois worklist can be found [Unnikrishnan et al. 2015]. A Collection based BFS implementation on GPU (written in Falcon) can be found in [Unnikrishnan et al. 2015].

---

5.4. Foreach Statement

Code generation for a foreach statement depends on the object on which it is called and where (GPU/CPU) the object is allocated. Nested parallelism using foreach is not supported. We convert inner foreach statements of nested foreach statements to simple for loop statements during code generation.

The outermost loop is retained as a foreach statement and is converted to a CUDA kernel call / OpenMP pragma (except for Collection on CPU) in the generated code. Figure 6 shows the code generated for the relaxgraph() function and its foreach call from Figure 1, with the target being GPU. Since foreach statement inside relaxgraph() is nested inside foreach statement from main(), the foreach inside relaxgraph() is converted to a simple for loop. The variable TPB (Threads Per Block) corresponds to (MaxThreadsPerBlock - MaxThreadsPerBlock % CoresPerSM) for the GPU device on which the CUDA kernel is being called. We also make sure that a kernel executes by splitting a kernel call into multiple calls, if the number of threads or blocks for the kernel call is above the allowed value for device. Each Edge in Falcon stores two values in the edges array, the
destination Point and weight of the Edge. When a program uses innbrs iterator and outnbrs iterators, the inedges arrays of the Graph class stores two fields: source Point of the incoming Edge and an index in to the edges array which can be used to find out weight of the incoming Edge, which is stored in edges arrays.

Figure 7 shows the code generated for relaxgraph() function and its foreach statement when SSSP is written for a multicore CPU. The variable TOT_CPU stores the number of CPU cores available. The MIN function is converted to GMIN for GPU and HMIN for CPU. This convention is used throughout Falcon, as can be seen with Graph type converted to HGraph or GGraph based on where it is allocated.

Falcon stores the beginning index of neighbors of a Point in the index field of Graph class and outdegree of the point is found by taking the difference of index value of the nextpoint and this point (Line 6, Figure 7). The foreach statement in relaxgraph() processes all the neighbors of a Point serially, using a simple for loop. Similar code is generated for other iterators of Point and Edge data type.

We have experimented with warp-based code generation as well. However, we find that performance improvement is not always positive across benchmarks. Details of warp-based code generation are provided in [Unnikrishnan et al. 2015].

```c
int <GPU> changed;
SSSPBFS(char *name) { //begin SSSPBFS
    Graph hgraph;//Graph object on CPU
    hgraph.addPointProperty(dist,int);
    hgraph.addProperty(changed,int);
    hgraph.getType() <GPU> graph0;//Graph on GPU0
    hgraph.getType() <GPU> graph1;//Graph on GPU1
    hgraph.addPointProperty(dist1,int);
    hgraph.read(name);//read Graph from file to CPU
    graph0=hgraph;//copy entire Graph to GPU0
    graph1=hgraph;//copy entire Graph to GPU1
    foreach(t In graph0.points)t.dist=1234567890;
    foreach(t In graph1.points)t.dist=1234567890;
    graph0.points[0].dist=0;
    graph1.points[0].dist=0;

    parallel sections { //do in parallel
        section {//compute BFS on GPU1
            while(1){
                graph1.changed[0]=0;
                foreach(t In graph1.points)
                    BFS(t,graph1);
                if(graph1.changed[0]==0) break;
            }
        }
        section {//compute SSSP on GPU0
            while(1){
                graph0.changed[0]=0;
                foreach(t In graph0.points)
                    SSSP(t,graph0);
                if(graph0.changed[0]==0) break;
            }
        }
    }
}
```

Fig. 8: Multi-GPU BFS and SSSP in Falcon.

### 5.5. parallel sections, Multiple GPUs and Multiple Graphs

Falcon supports concurrent kernel execution using parallel sections. Falcon also supports multiple GPUs and Graphs. When multiple GPUs are available and multiple GPU Graph objects exist in the input program, each Graph object will be assigned a GPU number in a round robin fashion by the Falcon compiler. A GPU is assigned more than one Graph object if the number of GPU Graph objects exceeds the total number of GPUs available. Falcon assumes that a Graph object fits completely within a single GPU and proceeds with code generation. If there is more than one GPU Graph object, object allocation and kernel calls will be preceded by a call to cudaSetDevice() function, with the GPU number assigned to the object as its argument. It is possible to execute either the same algorithm or different algorithms on the Graph objects in the various GPUs.

For parallel kernel execution on different GPUs, each foreach statement should be placed inside a different section of the parallel sections statement. The parallel sections
statement gets converted to a OpenMP parallel region pragma, which makes it possible for the code segments in different sections inside the parallel sections to run in parallel. The method that we use for assigning Graphs to different GPUs is not optimal and the search for a better one is part of future work. The code fragment in Figure 8 shows how SSSP and BFS are computed at the same time on different GPUs using a parallel sections statement of Falcon. An Important point to be noted here relates to how changed variable is used in the code. If we declare changed as shown in Line 1 of Figure 8, it will be allocated in GPU device 0. So, to ensure that changed appears in each device, it is added as a Graph Property in Line 5.

5.6. Inter-Device Communication

Copying data between the CPU and the GPU is translated to cudaMemcpy which has different forms for the various assignment statements in Falcon. When an entire property of Graph, say Point or Edge is copied from GPU or to GPU, a cudaMemcpy operation is called to transfer a block of data. Falcon allows direct usage of GPU variables of basic types such as int, bool etc. inside CPU code. These statements will be converted to cudaMemcpyFromSymbol(Line 30, Figure 1) and cudaMemcpyToSymbol(Line 28, Figure 1) for data transfer from GPU and to GPU respectively, using compiler generated temporary variables.

In the SSSP() example, dist property of all the points is copied by an assignment statement:

\[
\text{hgraph.dist} = \text{graph.dist}; \quad // \text{Line 33 of Figure 1.}
\]

The generated CUDA code for this statement is shown in Figure 9. The above statement needs two cudaMemcpy operations as graph.extra is a GPU location and we cannot access graph.extra.dist in cudaMemcpy, as this implies dereferencing a device location (something that cannot be done from the host). A programmer can use The GPU Graph object directly in the printf statement and Falcon compiler generates code to copy dist value of all points to temporary pointer variable and use that in printf statement.

Recent advances in GPU computing allow access to a unified memory across CPU and GPU (e.g., in CUDA 6.0 and Shared Virtual Memory in OpenCL 2.0 and AMD’s HSA architecture). Such a facility clearly improves programmability and considerably eases code generation. However, concluding about the performance effects of a unified memory would require detailed experimentation. For instance, CUDA’s unified memory uses pinning pages on the host. For large graph sizes, pinning of several pages would interfere with the host’s virtual memory processing, leading to reduced performance. We defer the use of unified memory in Falcon as a future work.

5.7. Synchronization statement

The single statement is used for synchronization in Falcon. The second variant of the single statement is needed in functions which make structural modifications to graphs (morph algorithms) and it requires a barrier for the entire function to be inserted automatically during code generation. The total number of threads inside a CUDA kernel with a grid barrier cannot exceed a value specific to the GPU device and so these functions run in such a way that one thread processes more than one element. Cautious functions need single to be called on a Collection before any modification to the elements of Collection and no new elements can be added to the same Collection after the single statement. The compiler performs this check and if this condition is violated the user is warned about possible incorrect results.
Falcon: A Graph Manipulation Language for Heterogeneous Systems

refine(Graph graph, triangle t) {
    Collection triangle[pred];
    if(t is a bad triangle and not deleted) {
        find the cavity of t(set of surrounding triangles)
        add all triangles in cavity to pred
    } //end single(pred){
        //statements to update cavity
    } else {
        //abort
    } //end if
}

Fig. 10: Usage of single statement in DMR(Pseudo code)

#define t ((struct struct_graph *)(graph.extra))
for(int i=0;i<pred.size;i++)
t->owner[pred.D_Vec[i]]=id;
gpu_barrier(++goal,arrayin,arrayout);//global barrier
for(int i=0;i<pred.size;i++) {//2nd attempt to lock
    if((t->owner[pred.D_Vec[i]]<id)
        break;//locked by lower thread,exit
    else if(t->owner[pred.D_Vec[i]]>id)
        t->owner[cav1]=id;//update lock with lower id
} //end for

int barrflag=0;
for(int i=0;i<pred.size;i++)
    if(t->owner[pred.D_Vec[i]]!=id){barrflag=1;break;}}
if(barrflag==0 ){ //update cavity }
else { //abort }

Fig. 11: Generated CUDA code

There is no support for a grid barrier in CUDA and we have implemented it as given in [Xiao and Chun Feng 2010]. The CPU code uses barrier provided by OpenMP. The way single statement is used in DMR is shown in Figure 10. Here pred is a Collection object which stores the set of all triangles in the cavity. If a lock is obtained on all the triangles then the cavity is updated else the corresponding thread is aborted.

Pseudo Code in Lines 5-9 Figure 10 get converted to the CUDA code shown in Figure 11. Both GPU and CPU versions follow the above code pattern, with appropriate GPU and CPU functions. We lock the triangles based on the thread id and if two or more cavities overlap only the thread with the lowest thread id will succeed in locking the cavity and others abort. The global barrier makes sure that the operations of all threads are complete up to the barrier before any thread can proceed. This generated code is similar to that used in LonestarGPU.

The first variant of single statement in Table III that locks a single object does not need a barrier. It uses the compare_and_swap variant of CUDA [Nickolls et al. 2008] and GCC [Stallman et al. 2011] for GPU and CPU respectively. This type of single statement is normally used in local computation algorithms such as MST computation. In order for the single to work properly, the property value must be reset to zero before entering the function in which single is executed.

5.8. Reduction Function

Reduction operation has been implemented on GPU objects. Translation of reduction functions to CUDA functions is straightforward [Harris 2007].

5.9. Modifying Graph Structure

Deletion of a graph element is by marking. Each point and edge has a boolean flag that marks its deletion status. We provide an interface that enables a programmer to check if an object has been deleted by another thread.

For adding a Point or an Edge we rely on atomics. For a Graph object with the name say graph, we add global variables falcgraphpoint, falcgraphedge which will be initialized to the number of points and edges in graph(resp.). When a programmer writes graph.addPoint in the Falcon program, that code will be replaced by a call to an automatically generated function falcadgraphpointfun(). This function atomically increments falcgraphpoint by one. Analogous functions exist for Edge and properties added using the addProperty function. Currently, none of properties (attributes) associated with graph elements can be auto-deleted (including the one added using

5.10. Heterogeneous Execution in SSSP optimization discussed in Section 4.

Automatic management of size is also needed for morph algorithms. For example in DMR, the Graph size increases and the pre-allocated memory may not be sufficient. A call to compiler generated realloc() function is inserted automatically after the code that modifies the Graph size. This realloc() function considers current size, the change in size and the available extra memory allocated and performs Graph reallocation, if necessary.

In general, graph algorithms exhibit both memory as well as control-flow irregularity. While Falcon does not try to remove any of them completely, it takes the following measures to achieve better coalescing and locality. (i) CSR representation enables accessing the nodes array in a coalesced fashion. It also helps achieve better locality as edges of a node are stored contiguously. (ii) Shared memory accesses for warp-based execution and reductions help improve memory latency. (iii) Optimized algorithms. Note that a high-level DSL allows us to tune an algorithm easily, such as the SSSP optimization discussed in Section 4.

5.10. Heterogeneous Execution in Falcon using Graph Partitioning

When a Graph object does not fit into GPU memory, the programmer can make use of the graph partitioning functions available in Falcon. Falcon currently supports partitioned execution with one CPU and multiple GPUs. Only Totem [Gharaibeh et al. 2013][Gharaibeh et al. 2012] supports partitioned execution. The partitioning algorithm, communication mechanism, and subgraph storage structures used in Falcon have been derived from Totem. But unlike Totem, Falcon hides all the internal details from the programmer. Falcon supports random partitioning, partitioning based on the degree of the nodes, and a new partitioning algorithm, called ordered partitioning. In this algorithm, if X and Y are the percentages (X+Y=100) of a graph to be allocated on two partitions, the first X% points and their edges are allocated on subgraph1, and the remaining graph on subgraph2 (similarly for partitioning with three or more subgraphs). We have tested partitioned execution only for vertex-centric algorithms (as in Totem). A non-vertex-centric algorithm requires edge-based processing and this may result in more communication, as the number of edges in a graph is usually much higher than the number of nodes. This will be explored in future work.

As in Totem, a node and all its edges are also stored in the same subgraph. If the destination node of an edge is in the other partition, it becomes a remote node. In the case of computation with GPU and CPU, new values of the remote nodes of a subgraph are sent to the other subgraph after the computation step, with the help of a communication buffer created in the CPU and the GPU. We support multi-GPU execution by enabling peeraccess between GPUs. The values are updated after each computation step for each subgraph in parallel without requiring any data transfer between GPUs. We have also implemented a basic version of partitioned execution using Unified Virtual Addressing (UVA), which is possible for Nvidia-GPUs with compute capability >=2.0. But computation with peeraccess is faster than with UVA.

A programmer is required to use the parallel foreach construct with the initial Graph object and the Falcon compiler automatically generates CUDA and OpenMP version codes for the GPU and the CPU (resp.). The compiler also determines the properties of a node (Point) that are updated in a parallel region. The programmer must specify a function for updating the values of properties of Points in the Graph object. On receiving the new values of properties of Points from another subgraph, the values are updated using this function (e.g., the minimum of the current value and the incoming value is taken in SSSP and BFS).

Falcon code in Figure 12 shows how SSSP computation can be performed on an input using both GPU and CPU. The makePartition function in Line 14 of Figure 12 partitions the graph into two parts, one each on CPU (argument 1) and GPU (argument 2) using the partition algorithm based on the degree of nodes in a graph (argument 3).

After a computation step, the current values of remote nodes are communicated to the partition in which the remote node is actually present. The updating function, updatePartition() (Line 22) applies the function fun1 (defined in Line 1 and specified as shown in Line 15) to update...
the value. The update function does not need atomic operations as each thread is accessing a different location. The Falcon compiler optimizes data transfers between partitions by sending the values of only the required properties to remote partitions (e.g. property values of \texttt{Point} incom, which are read in \texttt{fun1}, in Figure 12).

```
1 fun1(Point ori, Point incom){
  if(ori.dist > incom.dist)
    ori.dist=incom.dist
}

relaxgraph(Point p, HGraph hgraph){
  foreach(t in p.outnbrs)
    MIN(t.dist, p.dist+hgraph.getWeight(p,t),
    hgraph.changed[0]);
}

main(int argc, char *argv[]){
  HGraph hgraph;
  hgraph.addPointProperty(dist, int);
  hgraph.addProperty(changed, int);
  hgraph.read(argv[1]);
  hgraph.makePartition(1,1,SORT_BY_DEGREE);
  hgraph.updateFunction(fun1);
  foreach(t In hgraph.points) t.dist=1234567890;
  hgraph.points[0].dist=0;
  while(1){
    hgraph.changed[0]=0;
    foreach(t In hgraph.points)relaxgraph(t,hgraph);
    hgraph.updatePartition();
    if(hgraph.changed[0]==0)break; //end while
  }
  for(int i = 0;i < hgraph.npoints; i++)
    printf("%d", hgraph.points[i].dist);
}
```

Fig. 12: Partitioned SSSP algorithm (unoptimized)

For partitions in GPU and CPU, two \texttt{cudaMemcpy} operations are needed, one for each partition. The values are updated using a CUDA kernel call for the GPU and an \texttt{OpenMP} parallel loop for the CPU. Space allocation for various buffers and the generation of code for communication are handled automatically by the Falcon compiler. The property, \texttt{changed}, gets duplicated for each partition (also handled by the Falcon compiler). The \texttt{Graph} class contains pointers to the HGraph (GGraph ) class and these are used to allocate subgraphs on the CPU (GPU). The parallel call to \texttt{relaxgraph} gets converted to a CUDA kernel call and an \texttt{OpenMP} pragma for the GPU and the CPU, respectively. The if statement checks whether the value in the variable \texttt{changed} is unchanged (in both the partitions). If a programmer wants to execute only on multiple-GPUs or multiple-GPUs and CPU, the first two arguments are required to be modified. A programmer can also specify the percentage of a \texttt{Graph} object to be allocated on the CPU and GPUs using command line arguments.

The above example shows the ease of programming in Falcon using partitioned graphs. Falcon currently supports only vertex-centric algorithms and has been tested using a combination of multiple GPUs and a single CPU.

6. EXPERIMENTAL EVALUATION

To execute the CUDA codes, we have used an Nvidia multi-GPU system with Four GPUs (One Kepler K20c GPU with 2496 cores running at 706 MHz and 6 GB memory, two Tesla C2075 GPUs each with 448 cores running at 1.15 GHz and 6 GB memory, one Tesla C2050 GPU with 448 cores running at 1.15 GHz and 4 GB memory). Multicore codes were run on Intel(R) Xeon(R) E5645 CPU, with two hex-core processors (total 12 cores) running at 2.4 GHz with 24 GB memory. All the GPU codes were by default run on Kepler K20c (device 0). The CPU results are shown as speedup of 12-threaded codes against single-threaded Galois code. We used Ubuntu 14.04 server with g++-4.8 and CUDA-7.0 for compilation.

We compared the performance of the Falcon-generated CUDA code against LonestarGPU-2.0 and Totem [Gharaibeh et al. 2013][Gharaibeh et al. 2012], and the multicore code against that of Galois-2.2.1 [Pingali et al. 2011], Totem and GreenMarl [Hong et al. 2012]. LonestarGPU does not run on multi-core CPU and Galois has no implementation on GPU. Only Totem supports implementation of an algorithm on multiple GPUs using graph partitioning and Falcon’s comparison with Totem on this aspect is described in subsection 6.3.

Results are shown for three cautious morph algorithms (SP, DMR and dynamic SSSP) and three local computation algorithms (SSSP, BFS and MST). Falcon achieves close to 2\times and 5\times re-
Table VI. Inputs used for local computation algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Graph Type</th>
<th>Total Points</th>
<th>Total Edges</th>
<th>BFS distance</th>
<th>Max Nbrs</th>
<th>Min Nbrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand1</td>
<td>Random</td>
<td>16M</td>
<td>64M</td>
<td>20</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>rand2</td>
<td>Random</td>
<td>32M</td>
<td>128M</td>
<td>18</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>rmat1</td>
<td>Scale Free</td>
<td>10M</td>
<td>100M</td>
<td>INF</td>
<td>1873</td>
<td>0</td>
</tr>
<tr>
<td>rmat2</td>
<td>Scale Free</td>
<td>20M</td>
<td>200M</td>
<td>INF</td>
<td>2525</td>
<td>0</td>
</tr>
<tr>
<td>road1(usa-ctr)</td>
<td>Road Network</td>
<td>14M</td>
<td>34M</td>
<td>3826</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>road2(usa-full)</td>
<td>Road Network</td>
<td>23M</td>
<td>58M</td>
<td>6261</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Table VII. Lines of codes for algorithm in different frameworks / DSLs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Falcon CPU</th>
<th>Green-Marl</th>
<th>Galois</th>
<th>Totem CPU</th>
<th>Falcon GPU</th>
<th>Lonestar GPU</th>
<th>Totem GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>26</td>
<td>24</td>
<td>310</td>
<td>400</td>
<td>28</td>
<td>140</td>
<td>200</td>
</tr>
<tr>
<td>SSSP</td>
<td>35</td>
<td>24</td>
<td>310</td>
<td>60</td>
<td>38</td>
<td>170</td>
<td>330</td>
</tr>
<tr>
<td>MST</td>
<td>113</td>
<td>N.A.</td>
<td>590</td>
<td>N.A.</td>
<td>103</td>
<td>420</td>
<td>N.A.</td>
</tr>
<tr>
<td>DMR</td>
<td>302</td>
<td>N.A.</td>
<td>1011</td>
<td>N.A.</td>
<td>308</td>
<td>860</td>
<td>N.A.</td>
</tr>
<tr>
<td>SP</td>
<td>198</td>
<td>N.A.</td>
<td>401</td>
<td>N.A.</td>
<td>185</td>
<td>420</td>
<td>N.A.</td>
</tr>
<tr>
<td>Dynamic SSSP</td>
<td>51</td>
<td>N.A.</td>
<td>N.A.</td>
<td>N.A.</td>
<td>56</td>
<td>165</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Fig. 13: Speedup of SSSP, BFS and MST on GPU

duction in number of lines of code (see Table VII) for morph algorithms and local computation algorithms respectively compared to the hand-written code. We have measured the running time from the beginning of the computation phase till its end. This includes the cost of communication between the CPU and the GPU during this period. We have not included the running time for reading and copying the Graph object to the GPU and for copying results from the GPU. Absolute running times for all the algorithms can be found in [Unnikrishnan et al. 2015].

6.1. Local Computation Algorithms

Figure 13 shows the results for BFS, SSSP and MST on GPU and Figure 14 shows the results for BFS and SSSP on CPU. MST speedup on CPU is shown in Figure 15. We experimented with
several graph types (such as the Erdős-Rényi model random graphs [Erdős and Rényi 1960], road networks, and scale-free graphs) and have shown results for two representative graphs from each category, with several million edges. Details can be seen in Table VI. Road network graphs are real road networks of USA [DIMACS 2009], have less variance in degree distribution, but have large diameter. Scale-free graphs have been generated using GTGraph [Bader and Madduri 2006] tool, have a large variance in degree distribution but exhibit small-world property. Random graphs have been generated using the graph generation tool available in Galois.

SSSP. Results for SSSP on GPU have been plotted as speedup over best time reported by LonestarGPU variants (worklist based SSSP and Bellman-Ford style SSSP). We find that Falcon SSSP (Figure 1) is faster than LonestarGPU. This is due to the optimization used in the Falcon program using the uptd field, which eliminates many unwanted computations. For rmat2 input worklist based SSSP of LonestarGPU went out of memory and speedup shown is over slower Bellman-Ford style SSSP of LonestarGPU. The speedup for SSSP on GPU is shown for Totem and Falcon with respect to LonestarGPU in Figure 13(a).

The results for SSSP on CPU are plotted as speedup over Galois single threaded code (Figure 14(a)). Falcon and Galois use a Collection based \(\Delta\)-stepping implementation. Totem and GreenMarl do not have a \(\Delta\)-stepping implementation. Hence, Totem and GreenMarl are always slower than Galois and Falcon for road network inputs. GreenMarl failed to run on rmat input giving a runtime error on \texttt{std::vector::reverse}(). It is important to note that Bellman-Ford variant of the SSSP code (Figure 1) on CPU with 12 threads is about \(8\times\) slower than that of the same on GPU. It is the worklist based \(\Delta\)-stepping algorithm which made CPU code fast. BFS and MST also benefit considerably from worklist based execution on CPU.

BFS. Results for BFS on GPU are compared as speedup over the best running times reported by LonestarGPU. We took the best running times reported by worklist based BFS and Bellman-Ford variant BFS implementations. The worklist based BFS performed faster only for road network input. Falcon also has a worklist based BFS on GPU which is slower by about \(2\times\) compared to that of LonestarGPU. Totem framework is too slow on road network due to lack of worklist based implementation. GreenMarl failed to run on rmat input giving a runtime error on \texttt{std::vector::reverse}().

Falcon BFS code on CPU always outperformed Galois BFS, due to our optimizations (Figure 14(b)). Totem and GreenMarl are again slower on road inputs. Totem performed better than Falcon BFS on GPU for scale free graphs. Totem runs algorithms using graph partitioning which benefits graphs that follow the power law distribution, and rmat graphs do follow the power law [Gharaibeh et al. 2012]. The speedup for BFS on GPU is shown for Totem and Falcon with respect to LonestarGPU in Figure 13(b).

MST. LonestarGPU has a Union-Find based MST implementation. Falcon GPU code for MST always outperformed that of LonestarGPU for all inputs, with the help of better implementation of Union-Find that Falcon has for GPU. But our CPU code showed a slowdown compared to Galois (about \(2\times\) slowdown). Galois has a better Union-Find implementation based on object location as key. The Speedup for MST on GPU is shown in Figure 13(c) and same for CPU is shown in Figure 15.

Multi-GPU. Figure 16 shows the speedup of Falcon when algorithms BFS, SSSP and MST are executed on three different GPUs in parallel for the same input, when compared to their separate executions on the same GPU. One should not be confused with speedup values in Figure 16 and values in Figure 13, because for road networks, SSSP running time was very high compared to the MST running time, and for other inputs (random, rmat) MST running time was higher. It is also possible to run algorithms on CPU and GPU in parallel using the parallel sections statement. A Programmer can decide where to run a program by allocating a Graph object on GPU or CPU, which can be specified in a declaration statement with or without using \texttt{<GPU>} tag. He/She can then place appropriate \texttt{foreach} statements in each section of the parallel sections statement of Falcon. For example, SSSP on road network inputs can be run on CPU (because it is
6.2. Morph Algorithms

We have specified three morph algorithms using Falcon: DMR, SP and dynamic SSSP. All these algorithms have been implemented as cautious algorithms and we have compared the results with implementations using LonestarGPU and Galois (other frameworks do not support mutation of graphs). Other morph algorithms can be easily specified in Falcon.

Delaunay Mesh Refinement (DMR). DMR implementation in LonestarGPU relies on a global barrier, which can be implemented either by returning to the CPU and launching another kernel, or by emulating a grid-barrier in software [Xiao and chun Feng 2010]. LonestarGPU uses the latter slow on GPU) and for random and rmat graph inputs, on GPU. The effort required to modify codes for CPU or GPU is minimal with Falcon.

We have Falcon implementations of many other graph algorithms such as page ranking, betweenness centrality, etc., and these can be found in [Unnikrishnan et al. 2015]. We found it easy to implement such algorithms in Falcon without worrying about the details of the underlying architecture.
approach as it allows saving the state of the computation in local and shared memory across barriers inside the kernel (which is infeasible in the first approach where the kernel is terminated) and this approach is used in Falcon DSL code as well. Unfortunately, grid-level barriers pose a limit on the number of threads with which a kernel can be launched, as all the thread-blocks need to be resident and all the threads must participate in the barrier; otherwise, the kernel execution hangs. Therefore, both LonestarGPU and Falcon-generated code restrict the number of launched threads, thereby limiting parallelism. However, it avoids costly global memory access. This is also observable in other morph algorithm implementations needing a grid-barrier. Figure 17(a) and 17(b) show the performance comparison of DMR code for GPU and CPU on input meshes containing a large number of triangles in the range 0.5 to 10 million. Close to 50% of the triangles in each mesh are initially bad (that is, they need to be processed for refinement). Galois goes out of memory for 10 million triangles or more, and terminates. Falcon code is about 10% slower compared to LonestarGPU code and both used the same algorithm. This can be due to the inefficiency arising from conversion of DSL code to CUDA code. Speedup shown is for mesh refinement code (including communication involved during that time), after reading mesh.

Survey Propagation (SP). Survey Propagation algorithm [Braunstein et al. 2005] deletes a node when its associated probability becomes close to zero and this makes SP a morph algorithm. In this implementation, we implemented the global barrier on a GPU by returning to the CPU, as no local state information needs to be carried across kernels (the carried state of variables is stored in global memory). A similar approach is used in LonestarGPU as well.

The first four rows of Table VIII show how SP works for a clause(M)-to-literal(N) ratio of 4.2 and 3 literals-per-clause(K) for different input sizes and the last three rows are for different values for

<table>
<thead>
<tr>
<th>Input(K, N, M)</th>
<th>Galois</th>
<th>Falcon</th>
<th>LonestarGPU</th>
<th>Falcon GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,1x10^6, 4.2x10^6)</td>
<td>67</td>
<td>46</td>
<td>26</td>
<td>23</td>
</tr>
<tr>
<td>(3,2x10^6, 8.4x10^6)</td>
<td>147</td>
<td>76</td>
<td>55</td>
<td>47</td>
</tr>
<tr>
<td>(3,3x10^6, 12.6x10^6)</td>
<td>232</td>
<td>114</td>
<td>86</td>
<td>69</td>
</tr>
<tr>
<td>(3,4x10^6, 16.8x10^6)</td>
<td>322</td>
<td>147</td>
<td>117</td>
<td>93</td>
</tr>
<tr>
<td>(4,4x10^6, 9.9x10^6)</td>
<td>1867</td>
<td>149</td>
<td>118</td>
<td>95</td>
</tr>
<tr>
<td>(5,1x10^6, 21.1x10^6)</td>
<td>killed</td>
<td>356</td>
<td>414</td>
<td>314</td>
</tr>
<tr>
<td>(6,1x10^6, 43.4x10^6)</td>
<td>killed</td>
<td>1322</td>
<td>1180</td>
<td>928</td>
</tr>
</tbody>
</table>
the clause(M)-to-literal(N) ratio. We observe that Falcon-generated code always performs better than both multicore Galois with 12 threads and LonestarGPU. Note that performance has been compared with LonestarGPU-1.0 and Galois-2.1 codes. New versions of both these frameworks use a new algorithm, which is yet to be coded in Falcon. Multicore Galois goes out-of-memory for higher values of (K, N, M), whereas LonestarGPU and Falcon versions complete successfully. LonestarGPU allocates each property of clause and literal in separate arrays whereas in Falcon, each property of clause and literal is put in structures, one each for clause and literal. Galois has a worklist based implementation of the algorithm. Also both Galois and LonestarGPU work by adding edges from clauses (Point in Graph) to each literal (Point in Graph) in the clause. But Falcon takes a clause as an extra property of the Graph (like triangle was used in DMR) and that property stores literals (Points) of the clause in it. So our Graph does not have any explicit edges, and literals of a clause (which correspond to edges) can be accessed very efficiently from the clause property of the Graph. We find that Falcon code runs faster than that of both Galois and LonestarGPU. Writing an algorithm that maintains a clause as a property of a Graph in LonestarGPU and Galois is not an easy task.

Dynamic SSSP. In a dynamic Single Source Shortest Path (SSSP) algorithm, edges can be added or deleted dynamically. A dynamic algorithm where only edges get added (deleted) is called as an incremental (decremental) algorithm, whereas algorithms where both insertion and deletion of edges happen are called fully dynamic algorithms [Frigioni et al. 1998]. We have implemented an incremental dynamic algorithm on GPU and CPU using Falcon. We have used a variant of the algorithm by [Ramalingam and Reps 1996]. Insertions are carried out in chunks and then SSSP is (incrementally) recomputed. We found it difficult to add dynamic SSSP to the Galois system, because no Graph structure that allows efficient addition of big chunk of edges to an existing Graph object was found. LonestarGPU code has been modified to implement dynamic SSSP, and we compare it with our CPU and GPU versions. Falcon looks at functions used in programs that modify a Graph structure (addPoint(), addEdge(), etc.) and converts a Graph read() function in Falcon to the appropriate read() function of the HGraph class. For dynamic SSSP, the read() function allocates more space to add edges for each Point and makes the algorithm work faster. LonestarGPU code has also been modified in the same way. Results are shown in Figure 17(c), which shows the speedup of the incremental SSSP computation with respect to initial SSSP computation. SSSP on GPU is an optimized Bellman-Ford style algorithm that processes all the elements and so does many unwanted computations, while CPU code is \( \Delta \)-stepping algorithm. Implementation of a fully dynamic SSSP is easy in Falcon. Edge deletion is a harder problem and we do not deal with it.

6.3. Heterogeneous Execution with Graph Partitioning

Falcon supports execution of vertex-centric algorithms on CPU and multiple-GPUs using graph partitioning. We have collected results for two random graphs and three RMAT graphs. Random graphs are with 64M nodes (rand64) and 128M nodes (rand128) with number of total edges being four times the number of nodes. RMAT graphs are with 50M nodes (rmat50), 60M nodes (rmat60) and 80M nodes (rmat80) with total number of edges being ten times the number of nodes. Results are shown for SSSP and BFS on these inputs for execution on two GPUs (Figure 18(a)), and two GPUs and one CPU (Figure 18(b)) as compared to execution over single threaded CPU code. The reader should note that partitioned execution is to be used only when the graph does not fit into single GPU or single (multi-core) CPU memory. We utilized the GPU memory to the maximum possible extent for these large graphs. The rand128 input and rmat80 inputs did not fit in two GPUs and hence is executed on two GPUs and one CPU. The Totem framework and Falcon code were run on multi-GPU by enabling peeraccess and this is faster than code using Unified Virtual Addressing (UVA). The peeraccess method needs GPUs to be on the same I/O Hub and so we used two GPUs (Fermi C2075 and Fermi C2050) which are on the same I/O Hub in our multi-GPU machine. Totem needed recompilation for compute capability 2.0 and modification of code to assign GPU partitions to use devices with peeraccess. Our results were collected with ordered partitioning (because it worked
Fig. 18: Heterogeneous execution speedup comparison over single threaded CPU (SSSP,BFS).

better than other schemes with Falcon) and Totem uses random partitioning. Results are shown with time including partitioning time, execution time and communication time during computation.

7. CONCLUSION AND FUTURE WORK

We presented Falcon, a domain specific language for expressing graph algorithms. It supports writing explicitly parallel programs, thus retaining efficiency. By enabling an algorithmic specification at a higher level, it allows easy changes to the code and also its maintenance. Salient features of Falcon are that it supports morph algorithms, wherein the underlying graph structure may change and provides support for heterogeneous architecture, multi-GPU systems and multi-core CPUs. We illustrated its expressibility by generating CUDA and OpenMP code for morph algorithms such as Delaunay mesh refinement, survey propagation and dynamic SSSP. We showed that writing code for CPU and GPU are similar, except in the case where variables in GPU need to be annotated with <GPU> tag and we showed that the generated code performs close to (and sometimes better than) their hand-tuned implementations. We also presented preliminary results of execution of vertex-centric algorithms on partitioned graphs. In the future, the portability of Falcon will be improved by supporting OpenCL as the backend and by extending Falcon support for CPU Clusters. Automatic code generation without the programmer explicitly specifying the location of Graph objects and supporting speculation with rollback are also on the cards.

REFERENCES


Mark Harris. 2007. Optimizing Parallel Reduction in CUDA.


Shucai Xiao and Wu Chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. (April 2010).
