Falcon: A Graph Manipulation Language for Heterogeneous Systems

UNNIKRISHNAN CHERAMANGALATH, Department of CSA, Indian Institute of Science, Bangalore
RUPESH NASRE, Department of CSE, Indian Institute of Technology, Madras
Y. N. SRIKANT, Department of CSA, Indian Institute of Science, Bangalore

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 multicore 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 multicore CPUs. Using a set of benchmark graphs, we illustrate that the generated code performs close to the state-of-the-art hand-tuned implementations.

CCS Concepts: • **Software and its engineering → Compilers**

Additional Key Words and Phrases: Graph manipulation languages, domain specific languages, CUDA, OpenMP, GPU, multi-core CPU, morph algorithms, local computation algorithms

ACM Reference Format:
Unnikrishnan C, Rupesh Nasre, and Y. N. Srikant. 2015. Falcon: A graph manipulation language for heterogeneous systems. ACM Trans. Archit. Code Optim. 12, 4, Article 54 (December 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2842618

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 web graph, a community detection algorithm discovers likely communities in a social network, and 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 and parallelize because they are irregular.

New article, not an extension of a conference paper.
This work is supported by the IMPECS project of DST (Government of India) and MPI-SWS (Germany).
Authors' addresses: Unnikrishnan C and Y. N. Srikant, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India; emails: unni_c@csa.iisc.ernet.in; srikant@csa.iisc.ernet.in; R. Nasre, Department of Computer Science and Engineering, Indian Institute of Technology, Madras, 600036, India; email: rupesh@cse.iitm.ac.in.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1544-3566/2015/12-ART54 $15.00
DOI: http://dx.doi.org/10.1145/2842618

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

It would be helpful if a programmer could 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 to a manually parallelized hardware-centric code by an expert.

Our goal in this work is to bridge this performance gap between an autogenerated 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 handcrafted 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, whereas the second one allows our code generator to emit hardware-specific code.

Our specific contributions are as follows:

—The design of `Falcon`, a domain-specific language (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 CPUs, single GPUs, multi-GPUs, and heterogeneous backends. Our compiler supports worklist-based implementations of morph and local computation algorithms on the CPU that run much faster than most handwritten 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.

The rest of the article is organized as follows. Section 2 mentions the benefits of `Falcon`. Section 3 compares and contrasts the 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, whereas 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 GPUs, multi-GPUs, and a combination of CPUs and GPUs. 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 that are much faster than the state-of-the-art implementations (e.g., 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 the 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 GPUs and Galois [Pingali et al. 2011] for multicore CPUs, 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 of these on his or her own. Such a code is difficult to debug. This makes `Falcon` a new choice for coding parallel graph algorithms that is easy to use, easy to debug, and also efficient.

3. RELATED WORK

Green-Marl [Hong et al. 2012] and Elixir [Prountzos et al. 2012] are examples of graph DSLs, and both of them target multicore CPUs. 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 of 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 CPUs. Cautious morph algorithms have been implemented on the GPU by Nasre et al. [2013b]. 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 (BSP) 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 the 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 the GPU for speculative execution, and on misspeculation, that part of the code is executed on the CPU. An online profiling–based method by Kaleem et al. [2014] partitions work and distributes it across the CPU and GPU.

Table I. Related Works Comparison

References	A	B	C	D	E	F
Green-Marl [Hong et al. 2012], Elixir [Prountzos et al. 2012], [Hong et al. 2014]	√	x	x	√	x	x
[Ragan-Kelley et al. 2013]	√	x	x	√	√	x
Lonestar-GPU [Nasre et al. 2013b]	x	√	x	x	√	√
[Shun and Blelloch 2013; Roy et al. 2013; Zhang et al. 2015]	x	√	x	√	x	x
Medusa [Zhong and He 2014; Lee et al. 2009]	x	√	x	x	√	x
Totem [Gharaibeh et al. 2012, 2013]	x	√	x	√	√	x
Galois [Pingali et al. 2011]	x	√	x	√	x	√
[Burtscher and Pingali 2011; Sariyüce et al. 2013; Nasre et al. 2013a; Davidson et al. 2014; Khorasani et al. 2014; Mendez-Lojo et al. 2012; Prabhu et al. 2011; Harish and Narayanan 2007; Harish et al. 2009; Hong et al. 2011]	x	x	x	x	√	x
[Feng et al. 2012; Menon et al. 2012]	x	x	x	x	√	√
[Tian et al. 2008, 2011]	x	x	x	√	x	√
[Low et al. 2012; Bader and Madduri 2008; Gregor and Lumsdaine 2005]	x	x	√	√	x	x

Note: A, DSL; B, Framework; C, Library; D, CPU; E, GPU; F, Speculation.

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

The 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. [Feng et al. 2012] implemented methods for speculative parallelization of loops on the GPU that have irregular memory access and control flow. The CoRD [Tian et al. 2008, 2011] framework proposes methods for speculative execution on multicore CPUs. It supports rollbacks and morph algorithms that need not be cautious. More references related to graphs, graph DSLs, speculation, and so on, 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].

```
1  int <GPU> changed = 0; // Variable on GPU
2  relaxgraph(Point <GPU> p, Graph <GPU> graph) {
3      p.uptd = false;
4      foreach( t In p.outnbrs ){
5          MIN(t.dist, p.dist + graph.getWeight(p, t), changed);
6      }
7  }
8  reset(Point <GPU> t, Graph <GPU> graph) {
9      t.dist = t.olddist = 1234567890; t.uptd = false;
10 }
11 reset1(Point <GPU> t, Graph <GPU> graph) {
12     if (t.dist < t.olddist) t.uptd = true;
13     t.olddist = t.dist;
14 }
15 //main function on rhs
16 main(int argc, char *argv[]) {
17     Graph hgraph; // graph on CPU
18     hgraph.addPointProperty(dist, int);
19     hgraph.getType() <GPU> graph; // graph on GPU
20     graph.addPointProperty(uptd, bool);
21     graph.addPointProperty(olddist, int);
22     hgraph.read(argv[1]); // read graph on CPU
23     graph = hgraph; // copy graph to GPU
24     foreach (t In graph.points) reset(t,graph);
25     graph.points[0].dist = 0; // source has dist 0
26     graph.points[0].uptd = true;
27     while( 1 ){
28         changed = 0; //keep relaxing on GPU
29         foreach (t In graph.points) (t.uptd) relaxgraph(t,graph);
30         if (changed == 0) break; //reached fix point
31         foreach (t In graph.points) reset1(t,graph);
32     }
33     hgraph.dist = graph.dist; // copy all points dist to CPU
34     for (int i = 0; i < hgraph.npoints; ++i)
35             printf("i=%d dist=%d\n", i, hgraph.points[i].dist);
36 }
```

Fig. 1. Optimized GPU SSSP code in `Falcon`.

4.2. Example: Shortest Path Computation

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 the GPU. The algorithm first initializes the *dist* variable of all 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)) $dist[v] > dist[u] + weight(u, v)$. 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 through 32).

`Falcon` needs each variable that resides on the GPU to have the <`GPU`> tag preceding the variable name in the declaration statement (lines 1 and 9). Being a graph DSL, the type `Graph` is directly available in the language.

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 that 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 and 21 add two properties (*uptd, olddist*) to points in the `GPU Graph` object *graph*. Lines 22 and 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.

Table II. Data Types in `Falcon`

Data Type	Description	Major Fields	Major Functions
`Point`	Point in graph	x, y, z	del(), getNeighbors()
`Edge`	Edge in graph	src, dst, weight	del()
`Graph`	Entire graph	points[], edges[], npoints, nedges	addEdge(), addPoint(), getWeight(), read(), addEdgePropery(), sortEdges(), addProperty(), makePartition(), updatePartition()
`Set`	A static collection	size, parent	find(), union(), clear()
`Collection`	A dynamic collection	size	add(), del(), orderByIntValue(), clear()

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 the *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()* 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). Thus, whenever there is a reduction in the value of *dist* for even one `Point`, the variable *changed* is set to 1. Line 3 makes the *uptd* property of each `Point` whose current value is `true` to `false`. After each call to *relaxgraph()*, the *reset1()* function makes *uptd* `true` only for points whose distance from the source node was reduced in the last invocation of the *relaxgraph()* function (line 31). The variable *changed* is reset to zero before *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 `<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 `<GPU>` tag. Lines 20 and 21 should be modified to add the properties to the `CPU` graph object *hgraph*. There is no need to create a `GPU` graph object, and we should replace all occurrences of the `GPU` graph object *graph* with the `CPU` graph object *hgraph*. Lines 19, 23, and 33 will be absent in the 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.

```
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
}
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
    satisfies this condition.
}
//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.

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 of 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 `addEdgePropery()` is used to add a property to each edge in a `Graph` object with the same syntax as `addPointProperty()` used in line 18 of Figure 1. The `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 that are not necessarily direct functions of points and edges. For instance, such a function is used in DMR [Chew 1993] code, as the graph consists of a collection of *triangle*s, each *triangle* with three `Points` and a few extra properties. The statement shown next illustrates the way DMR code uses this function for a `Graph` object, *hgraph*.

hgraph.`addProperty`(*triangle*, `struct` *node*);

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

4.3.3. Set. A `Set` is an aggregate of unique elements (a set of threads, a set of nodes, etc.). A `Set` has a maximum size and cannot 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 minimum spanning tree (MST) algorithm [Chung and Condon 1996]. The way in which `Set` data type is declared in MST code is shown in Figure 3.

Line 2 declares objects of `Set` data type one each on the CPU and GPU. Each `Set` object contains a set of all 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

```
Graph hgraph, <GPU> graph;
Set hset[Point(hgraph)], set[Point(graph)];
```

Fig. 3. Use of `Set` in MST computation.

of the `Edge` are unioned into a single `Set`. The algorithm terminates when the `Set` has a single representative (assuming that the graph is connected) or when no edges are added to the MST in an iteration (for a disconnected graph). We mark all 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.

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` on which to operate, the underlying `Graph` object on the GPU, and a `Set` of points. Line 10 takes each outgoing neighbor of `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 that 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 [Hoberock and Bell 2011] for the GPU and the Galois worklist and its runtime for the multicore CPU. The use of a Galois worklist for the 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.

DMR [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 next with `Point` variables `P0` and `P1` (`Edge` declarations are similar). Given a `Graph` object g, we say that g is the `parent` of the points and edges in g.

`Point` P1, (*graph*)*P0*; //parent `Graph` of P0 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 preceding example, P0 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., P1 in the preceding example).

`Falcon` allows a programmer to specify on which GPU device the variable needs to be allocated with the optional integer argument along with the `<GPU>` tag. `Falcon` has a new keyword named `struct_rec`, which is used to declare recursive data structures.

Table III. Single Statement in `Falcon`

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

Table IV. `Foreach` Statement in `Falcon`

foreach(item (advance_expression) **In** object.**iterator**) (condition) {block of code}	Used for **Point**, **Edge** and **Graph** objects
foreach(item (advance_expression) **In** object) (condition) {block of code}	Used for **Collection** and **Set** objects

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 the `single` statement to lock a set of graph elements, as discussed later in this section.

When compared to other synchronization constructs such as the `synchronized` construct of Java or `lock` primitives in the 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 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, and 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 shared data (e.g., a set of neighboring nodes) are locked before proceeding with the computation. A thread succeeds if all 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 variants, the thread that succeeds in acquiring a lock executes the code following it, and if the optional `else` block is present, all threads that do not acquire the lock execute the code inside the `else` block.

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 variants. The 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, respectively) 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` statement (see Table IV) can also be a dereference of a pointer to an object. For examples on the 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 the `foreach` statement for different `Falcon` data types are shown in Table V.

Table V. Iterators for `Foreach` Statement in `Falcon`

Data Type	Iterator	Description
`Graph`	points	Iterate over all points in graph
`Graph`	edges	Iterate over all edges in graph
`Graph`	pptyname	Iterate over all elements in new ppty
`Point`	nbrs	Iterate over all neighboring points
`Point`	innbrs	Iterate over src point of incoming edges (directed `Graph`)
`Point`	outnbrs	Iterate over dst point of outgoing edges (directed `Graph`)
`Edge`	nbrs	Iterate over neighbor edges
`Edge`	nbr1	Iterate over neighbor edges of Point P1 in Edge(P1,P2) (directed `Graph`)
`Edge`	nbr2	Iterate over neighbor edges of Point P2 in Edge(P1,P2) (directed `Graph`)

A `foreach` statement gets converted to a CUDA kernel call or an `OpenMP pragma` (except for `Collection`) based on the object on which it is called: either a GPU object or a CPU object.

In a `Graph`, we can process all points and edges in parallel. An iterator called `pptyname` is generated automatically when a new property is added to a `Graph` object using the `addProperty()` function. This is often used in morph algorithms. When a property *triangle* is added to a `Graph` object using `addProperty()`, it generates an iterator *triangle*. There is no nested parallelism in our language. A nested `foreach` statement is converted to simple nested `for` loops in the generated code, except for the outermost `foreach` that is executed in parallel. The outermost `foreach` statement (executed in parallel) has an implicit global barrier after it (in the generated code).

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 region. 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 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, and so on, which are abstraction over the similar one in CUDA [Nickolls et al. 2008] and GCC [Stallman et al. 2011]. The MIN atomic function was used in Figure 1. We also provide a *barrier()* function that acts as a barrier for the entire group of threads in a CUDA kernel and `OpenMP` parallel region. A *genericbarrier()* that 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 BSP model of execution [Valiant 1990].

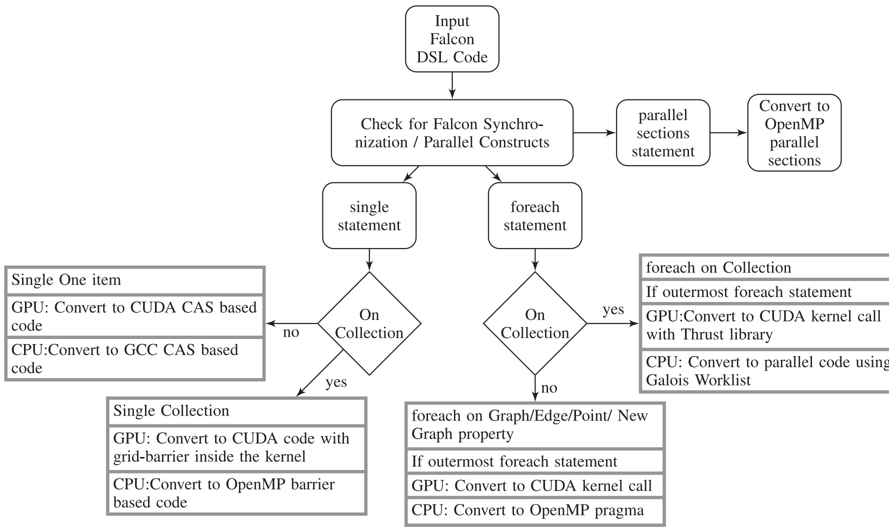


Fig. 4. `Falcon` code generation overview for parallelization and synchronization constructs.

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 CPU, respectively, and both inherit from a parent Graph class. The Graph class has an *extra* field (of type `void *`) that stores all 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 *ipe* and *fpe*, respectively. The generated code is compiled with *nvcc* and *g++*. Figure 4 gives an overview of how parallelization and synchronization is done for the CPU and GPU. The Falcon compiler names for all data types and functions specific to the 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 (e.g., `Collection`).

5.2. Properties

`Point` and `Edge` are converted to integer IDs. All extra properties of a `Graph` object are stored in the *extra* field and can be typecasted 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 the `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 objects created by the `getType()` compile-time function. In the SSSP example (Figure 1), Graphs on the

#define ep (struct struct_hgraph)	**alloc_extra_graph**(GGraph &graph) **{**
#define DH cudaMemcpyDeviceToHost	MA((void **) &(graph.extra), sizeof (ep));
#define HD cudaMemcpyHostToDevice	MC(&tmp, (ep *)(graph.extra), sizeof (ep),DH);
#define MA cudaMalloc	MA((void **) &(tmp.dist), sizeof (int)* graph.npoints);
#define MC cudaMemcpy	MA((void **) &(tmp.olddist), sizeof (int)* graph.npoints);
struct struct_hgraph { int *dist, *olddist; bool *uptd;};	MA((void **) &(tmp.uptd), sizeof (bool)* graph.npoints);
struct struct_hgraph tmp;	MC(graph.extra, &tmp, sizeof(ep), HD);}

Fig. 5. Allocating extra property for the `Graph` object on the GPU.

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 the *dist* field is allocated using *malloc()*, as *olddist and uptd* fields are associated only with the `GPU Graph` object (*graph*). Such simple optimizations are performed during the storage allocation phase of the `Falcon` compiler.

5.3. Set and Collection

The `Falcon` compiler has two C++ classes, HSet and GSet, which implement the CPU and GPU `Set` data types, respectively. 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 the GPU) and the Galois worklist along with its runtime code (for the 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 the 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 the 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 the GPU. Since the `foreach` statement inside *relaxgraph()* is nested inside the `foreach` statement from *main()*, the `foreach` inside *r*elaxgraph() is converted to a simple for loop. The variable threads per block (TPB) 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`

```
#define t (((struct struct_hgraph *)(graph.extra)))
__global__ void relaxgraph(GGraph graph, int x) {
    int id = blockIdx.x * blockDim.x +
                threadIdx.x + x;
    if (id <graph.npoints){
    if (t->uptd[id] == true){
    t->uptd=false;int falcft0 = graph.index[id];
    int falcft1 = graph.index[id+1]-graph.index[id];
    for (int falcft2 = 0; falcft2 < falcft1; falcft2++) {
        int ut0 = 2 * (falcft0 + falcft2); //edge index
        int ut1 = graph.edges[ut0].ipe; //dest point
        int ut2 = graph.edges[ut0 + 1].ipe;//weight
        GMIN(&t->dist[ut1], t->dist[id] + ut2
                    , changed);
    }
    } }
}
int flcBlocks=(graph.npoints/TPB+1)>MAXBLKS
        ?MAXBLCKS:graph.npoints/TPB+1);
for(int kk=0;kk<graph.npoints;kk+=TPB*flcBlocks)
relaxgraph < < < flcBlocks,TPB > > >(graph, kk);
```

Fig. 6. Code generated for the GPU SSSP relaxgraph() and its call.

```
1  #define t (((struct struct_hgraph
   *)(graph.extra)))
2  void relaxgraph(int &p ,HGraph &graph) {
       if (t->uptd[p] == true ){
       t->uptd=false;
       int falcft0 = graph.index[id];
6      int falcft1 =
       graph.index[id+1]-graph.index[id];
7      for (int falcft2 = 0; falcft2 < falcft1;
       falcft2++) {
           int ut0 = 2 * (falcft0 + falcft2);
           int ut1 = graph.edges[ut0].ipe;
           int ut2 = graph.edges[ut0 + 1].ipe;
           HMIN(&t->dist[ut1], t->dist[p] + ut2,
           ut1, changed);
12     }
       }
14 }
   #pragma omp parallel for
   for (int i = 0; i < hgraph.npoints; i++)
       relaxgraph(i, hgraph);
```

Fig. 7. Code generated for the CPU SSSP relaxgraph().

class stores two fields: source `Point` of the incoming `Edge` and an index in to the *edges* array that can be used to find out the weight of the incoming `Edge`, which is stored in *edges* arrays.

Figure 7 shows the code generated for the *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 the GPU and *HMIN* for the 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 the `Graph` class, and the *outdegree* of the point is found by taking the difference of the *index* value of the nextpoint and this point (see Figure 7, line 6). The `foreach` statement in relaxgraph() processes all 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].

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 the *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 various GPUs.

```
1  int <GPU> changed;
   SSSPBFS(char *name) { //begin SSSPBFS
     Graph hgraph;//Graph object on CPU
     hgraph.addPointProperty(dist,int);
5    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;
       }
     }
31 }//end SSSPBFS
```

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

```
#define ep (struct struct_hgraph)
#define DH cudaMemcpyDeviceToHost
#define HD cudaMemcpyHostToDevice
#define MC cudaMemcpy
struct struct_hgraph temp3;
MC(&temp3, (ep *)(graph.extra), sizeof(ep), DH);
MC(((ep *)(hgraph.extra))->dist, temp3.dist,
sizeof(int) * hgraph.npoints, DH);
```

Fig. 9. Code generated for line 34 in Figure 1.

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 the *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. Thus, 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 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 the GPU or to the 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` and `bool`, inside the CPU code. These statements will be converted to cudaMemcpyFromSymbol (see Figure 1, line 30) and cudaMemcpyToSymbol (see Figure 1, line 28) for data transfer from the GPU and to the GPU, respectively, using compiler-generated temporary variables.

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

hgraph.dist = graph.dist; // (see Figure 1, line 33)

The generated CUDA code for this statement is shown in Figure 9. The preceding statement needs two cudaMemcpy operations, as *graph.extra* is a GPU location, and we

```
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
5   single(pred){
6     //statements to update cavity
7   }else {
8       //abort
9    }//end single
  }//end if
}//end refine
```

Fig. 10. Usage of single statement in DMR (pseudocode).

```
#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
gpu_barrier(++goal,arrayin,arrayout);//global barrier
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.

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 the `Falcon` compiler generates code to copy the *dist* value of all points to a temporary pointer variable and use that in *printf* statement.

Recent advances in GPU computing allow access to a unified memory across the 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 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 that 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, 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.

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

Pseudocode in lines 5 through 9 in Figure 10 get converted to the CUDA code shown in Figure 11. Both GPU and CPU versions follow the preceding code pattern, with appropriate GPU and CPU functions. We lock the *triangle*s 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 the `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 the GPU and CPU, respectively. This type of `single` statement is normally used in local computation algorithms such as MST computation. For `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 of, say, *graph*, we add global variables *falcgraphpoint*, *falcgraphedge*, which will be initialized to the number of points and edges in *graph*, respectively. When a programmer writes *graph*.`addPoint` in the `Falcon` program, that code will be replaced by a call to an automatically generated function *falcaddgraphpointfun*(). 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 autodeleted (including the one added using addProperty); their deletion must be explicitly coded by the programmer. DMR deletes *triangle*s by storing a Boolean flag in the property *triangle* and making that flag value `true` for deleted triangles.

Automatic management of size is also needed for morph algorithms. For example, in DMR, the `Graph` size increases and the preallocated memory may not be sufficient. A call to the 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 and control-flow irregularity. Although 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, and 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 to improve memory latency; and (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 the 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. 2012, 2013] supports partitioned execution. The partitioning algorithm, communication mechanism, and subgraph storage structures used in `Falcon` have been derived from

```
1  fun1(Point ori, Point incom){
       if(orig.dist > incom.dist)
           orig.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);
13     hgraph.read(argv[1]);
14     hgraph.makePartition(1,1,SORT_BY_DEGREE);
15     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);
22       hgraph.updatePartition();
         if(hgraph.changed[0]==0)break; }//end while
       for(int i = 0;i < hgraph.npoints; i++)
         printf("%d", hgraph.points[i].dist);
   }//end main
```

Fig. 12. Partitioned SSSP algorithm (unoptimized).

Totem. But unlike Totem, `Falcon` hides all 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 of 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 the 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 of 2.0 or greater. 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 the `OpenMP` version codes for the GPU and the CPU, respectively. 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 the GPU and CPU. The *makePartition* function in line 14 of Figure 12 partitions the graph into two parts, one each on the 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

Table VI. Inputs Used for Local Computation Algorithms

Input	Graph Type	Total Points	Total Edges	BFS Distance	Maximum Neighbors	Minimum Neighbors
rand1	Random	16M	64M	20	17	1
rand2	Random	32M	128M	18	17	1
rmat1	Scale Free	10M	100M	INF	1,873	0
rmat2	Scale Free	20M	200M	INF	2,525	0
road1(usa-ctr)	Road Network	14M	34M	3,826	9	1
road2(usa-full)	Road Network	23M	58M	6,261	9	1

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 `Point` incom, which are read in fun1, in Figure 12).

For partitions in the GPU and CPU, two `cudaMemcpy` operations are needed, one for each partition. The values are updated using a CUDA kernel call for the GPU and an `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 *changed* gets duplicated for each partition (also handled by the `Falcon` compiler). The `Graph` class contains pointers to the HGraph (GGraph) class, and these are used to allocate subgraphs on the CPU (GPU). The parallel call to *relaxgraph* gets converted to a CUDA kernel call and an `OpenMP` pragma for the GPU and CPU, respectively. The `if` statement checks whether the value in the variable *changed* is unchanged (in both 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 `Graph` object to be allocated on the CPU and GPUs using command line arguments.

The preceding 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

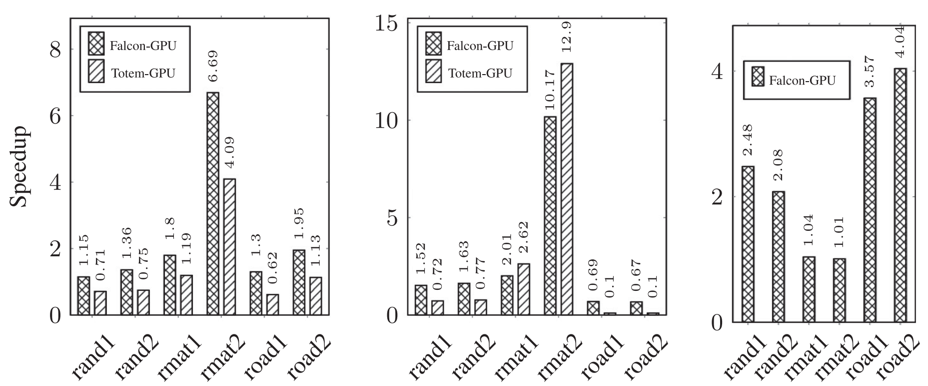
To execute the CUDA codes, we have used an Nvidia multi-GPU system with four GPUs (one Kepler K20c GPU with 2,496 cores running at 706MHz and 6GB memory, two Tesla C2075 GPUs each with 448 cores running at 1.15GHz and 6GB memory, and one Tesla C2050 GPU with 448 cores running at 1.15GHz and 4GB memory). Multicore codes were run on an Intel Xeon E5645 CPU, with two hex-core processors (total 12 cores) running at 2.4GHz with 24GB memory. All GPU codes were by default run on a Kepler K20c (device 0). The CPU results are shown as speedup of 12-threaded codes against single-threaded Galois code. We used an 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. 2012, 2013], and the multicore code against that of Galois-2.2.1 [Pingali et al. 2011], Totem, and Green-Marl [Hong et al. 2012]. LonestarGPU does not run on a multicore CPU, and Galois has no implementation on a 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 Section 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$\times$ reduction in number of lines of code (see Table VII) for morph algorithms and local computation algorithms, respectively, compared to the handwritten

Table VII. Lines of Codes for Algorithm in Different Frameworks/DSLs

Algorithm	Falcon CPU	Green-Marl	Galois	Totem CPU	Falcon GPU	Lonestar GPU	Totem GPU
BFS	26	24	310	400	28	140	200
SSSP	35	24	310	60	38	170	330
MST	113	NA	590	NA	103	420	NA
DMR	302	NA	1,011	NA	308	860	NA
SP	198	NA	401	NA	185	420	NA
Dynamic SSSP	51	NA	NA	NA	56	165	NA



(a) SSSP speedup over LonestarGPU (b) BFS speedup over LonestarGPU (c) MST speedup over LonestarGPU

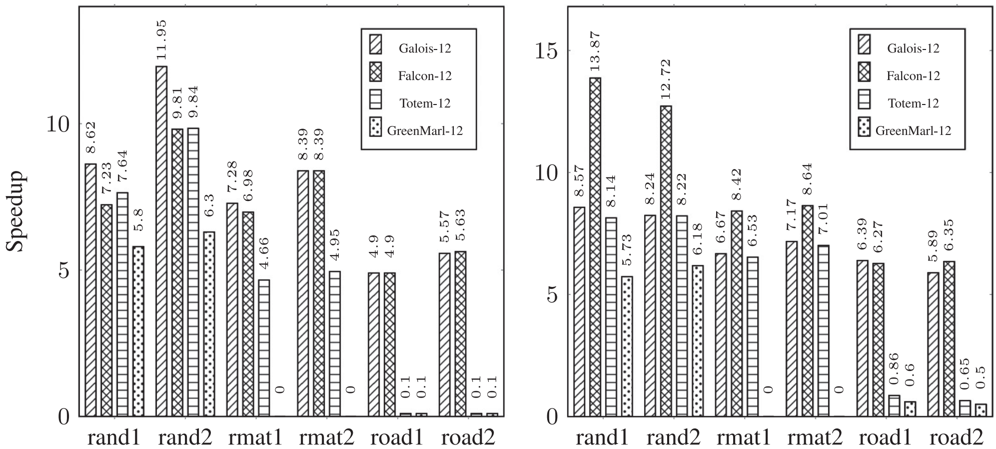
Fig. 13. Speedup of SSSP, BFS, and MST on the GPU.

code. We have measured the running time from the beginning of the computation phase until 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 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 the GPU, and Figure 14 shows the results for BFS and SSSP on the CPU. MST speedup on the CPU is shown in Figure 15. We experimented with several graph types (e.g., the Erdös-Rényi model random graphs [Erdös 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 the United States [DIMACS 2009] and have less variance in degree distribution but large diameter. Scale-free graphs have been generated using the GTGraph [Bader and Madduri 2006] tool and 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.

Single source shortest path. Results for SSSP on the GPU have been plotted as speedup over the 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 the slower



(a) SSSP speedup over Galois single

(b) BFS speedup over Galois single

Fig. 14. Speedup of SSSP and BFS on the CPU.

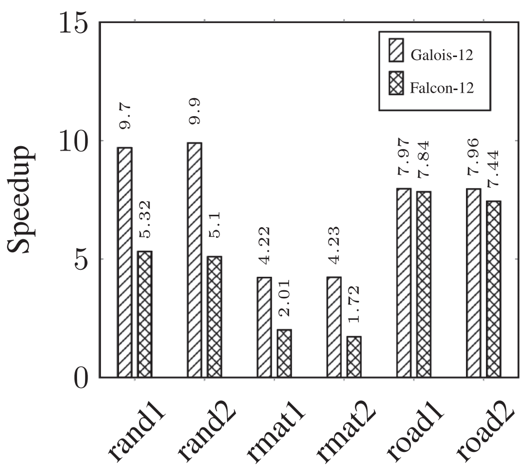


Fig. 15. Speedup of MST on the CPU over the Galois single.

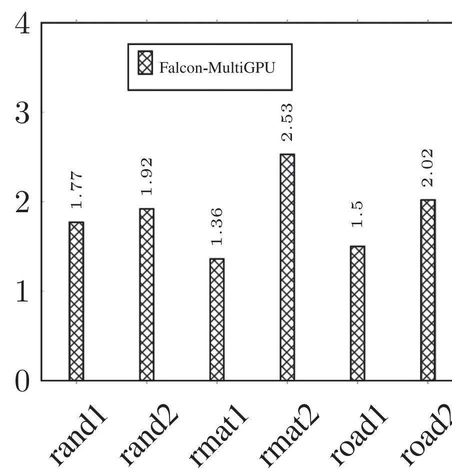


Fig. 16. Speedup of `Falcon` on a multi-GPU.

Bellman-Ford–style SSSP of LonestarGPU. The speedup for SSSP on the GPU is shown for Totem and `Falcon` with respect to LonestarGPU in Figure 13(a).

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

Breadth-first search. 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× compared to that of LonestarGPU. Totem framework is too slow on road network due to lack of worklist based implementation. Green-Marl failed to run on rmat input giving a runtime error on `std::vector::reverse()`.

`Falcon` BFS code on `CPU` always outperformed Galois BFS, due to our optimizations (Figure 14(b)). Totem and Green-Marl 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).

Minimum spanning tree. 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 the GPU. But our CPU code showed a slowdown compared to Galois (about 2× slowdown). Galois has a better Union-Find implementation based on object location as key. The speedup for MST on the GPU is shown in Figure 13(c) and the same for the 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 the 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 the GPU or CPU, which can be specified in a declaration statement with or without using the `<GPU>` tag. He or she can then place appropriate `foreach` statements in each `section` of the `parallel sections` statement of `Falcon`. For example, SSSP on road network inputs can be run on the CPU (because it is slow on the GPU), and for random and rmat graph inputs on the GPU. The effort required to modify codes for the CPU or GPU is minimal with `Falcon`.

We have `Falcon` implementations of many other graph algorithms, such as page ranking and betweenness centrality, 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.

6.2. Morph Algorithms

We have specified three morph algorithms using `Falcon`: DMR, SP, and dynamic SSSP. All of 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 easily be specified in Falcon.

Delaunay mesh refinement. 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 Feng 2010]. LonestarGPU uses the latter approach, as it allows saving the state of the

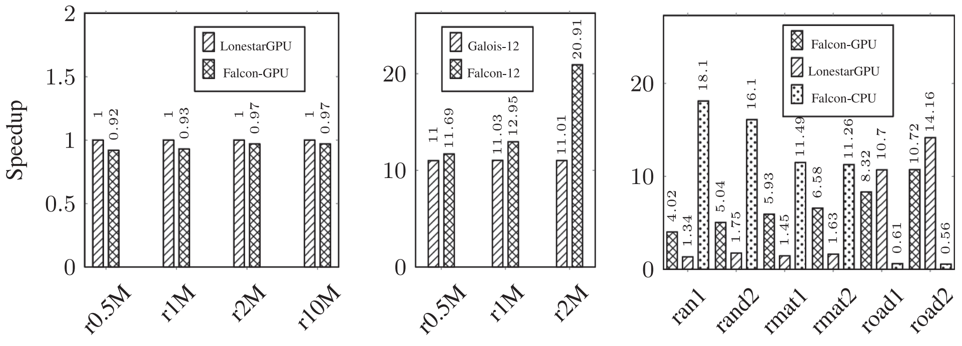
Table VIII. Performance Comparison for SP (Running Time in Seconds)

Input (K, N, M)	Galois (12 Threads)	Falcon (12 Threads)	Lonestar GPU	Falcon GPU
($3, 1x10^6$, $4.2x10^6$)	67	46	26	23
($3, 2x10^6$, $8.4x10^6$)	147	76	55	47
($3, 3x10^6$, $12.6x10^6$)	232	114	86	69
($3, 4x10^6$, $16.8x10^6$)	322	147	117	93
($4, 4x10^6$, $9.9x10^6$)	1867	149	118	95
($5, 1x10^6$, $21.1x10^6$)	Killed	356	414	314
($6, 1x10^6$, $43.4x10^6$)	Killed	1,322	1,180	928

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 thread blocks need to be resident and all 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 (b) show the performance comparison of DMR code for the GPU and CPU on input meshes containing a large number of triangles in the range of 0.5 to 10 million. Close to 50% of the triangles in each mesh are initially bad (i.e., 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. The SP 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 Lonestar GPU 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 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 to LonestarGPU-1.0 and Galois-2.1 codes. New versions of both of 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. In addition, 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. Thus, 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



(a) DMR speedup over LonestarGPU (b) DMR speedup over Galois single (c) Dynamic SSSP: Incremental speedup

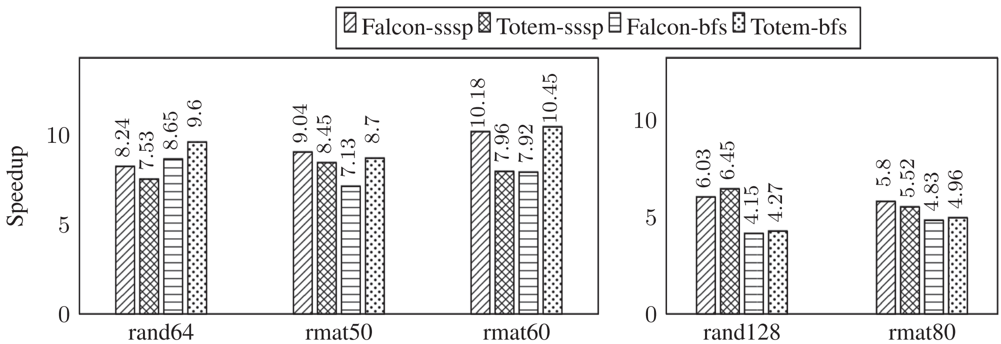
Fig. 17. Morph algorithm results: DMR and Dynamic SSSP.

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 single source shortest path. In a dynamic SSSP algorithm, edges can be added or deleted dynamically. A dynamic algorithm where only edges get added (deleted) is called 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 the GPU and CPU using `Falcon`. We have used a variant of the algorithm by Ramalingam and Reps [1996]. Insertions are carried out in chunks, 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 to 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 the GPU is an optimized Bellman-Ford–style algorithm that processes all of the elements and does many unwanted computations, whereas the CPU code is a Δ-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 the 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 the number of total edges being 4 times the number of nodes. RMAT graphs are with 50M nodes (rmat50), 60M nodes (rmat60), and 80M nodes (rmat80) with the total number of edges being 10 times the number of nodes. Results are shown for SSSP and BFS on these inputs for execution on two GPUs (Figure 18(a)), and 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 (multicore) CPU memory. We utilized the GPU memory



(a) Speedup on two GPUs (b) Speedup on two GPUs + one CPU

Fig. 18. Heterogeneous execution speedup comparison over a single-threaded CPU (SSSP, BFS).

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 a multi-GPU by enabling *peeraccess*, and this is faster than code using UVA. The *peeraccess* method needs GPUs to be on the same I/O hub, so we used two GPUs (Fermi C2075 and Fermi C2050) that 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 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 have presented `Falcon`, a DSL 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 DMR, SP, and dynamic SSSP. We showed that writing code for the CPU and GPU are similar, except in the case where variables in the GPU need to be annotated with a `<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 in the cards.

REFERENCES

D. Bader and K. Madduri. 2006. GTgraph: A Suite of Synthetic Graph Generators. Retrieved November 18, 2015, from http://www.cse.psu.edu/~madduri/software/GTgraph.

D. Bader and K. Madduri. 2008. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*.

David A. Bader and Kamesh Madduri. 2005. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *High Performance Computing—PiPC 2005*. Lecture Notes in Computer Science, Vol. 3769. Springer, 465–476.

A. Braunstein, M. Mézard, and R. Zecchina. 2005. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms* 27, 2, 201–226.

Martin Burtscher and Keshav Pingali. 2011. CUDA implementation of the tree-based Barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 75–92. http://iss.ices.utexas.edu/Publications/Papers/burtscher11.pdf.

Unnikrishnan C, R. Nasre, and Y. N. Srikant. 2015. *Falcon: A Graph Manipulation Language for Heterogeneous Systems*. Technical Report. CSA, IISc, Bangalore, India. http://www.csa.iisc.ernet.in/TR/2015/5/.

L. Paul Chew. 1993. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ACM Symposium on Computational Geometry*. 274–280.

S. Chung and A. Condon. 1996. Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm Retrieved November 18, 2015, from http://www.cs.ubc.ca/∼condon/papers/chungcondon96.pdf.

A. Davidson, S. Baxter, M. Garland, and J. D. Owens. 2014. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*. 349–359.

DIMACS. 2009. 9th DIMACS Implementation Challenge. Retrieved November 18, 2015, from http://www.dis.uniroma1.it/challenge9/download.shtml.

P. Erdős and A Rényi. 1960. On the Evolution of Random Graphs. Retrieved November 18, 2015, from http://www.renyi.hu/∼p_erdos/1960-10.pdf.

Min Feng, Rajiv Gupta, and Laxmi N. Bhuyan. 2012. Speculative parallelization on GPGPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 293–294.

Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasqualone. 1998. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *ACM Journal of Experimental Algorithmics* 3, Article No. 5.

Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 345–354.

Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. 2013. The energy case for graph processing on hybrid CPU and GPU systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*. ACM, New York, NY, Article No. 2.

Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. In *Proceedings of the Conference on Parallel Object-Oriented Scientific Computing (POOSC'05)*.

P. Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. 197–208.

P. Harish, V. Vineet, and P. J. Narayanan. 2009. *Large Graph Algorithms for Massively Multithreaded Architectures*. Technical Report IIIT/TR/2009/74. International Institute of Information Technology, Hyderabad, India.

Mark Harris. 2007. Optimizing Parallel Reduction in CUDA. Retrieved November 18, 2015, from http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

Jared Hoberock and Nathan Bell. 2011. *Thrust: A Productivity-Oriented Library for CUDA*. Technical Report. Nvidia Corporation.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 349–362.

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. *ACM SIGPLAN Notices* 46, 8, 267–276.

Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. 2014. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY, 208.

Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 151–162.

Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, NY, 239–252.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *ACM SIGPLAN Notices* 44, 4, 101–110.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8, 716–727.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, 135–145.

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 107–116.

Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception support and speculative execution on GPUs. *ACM SIGARCH Computer Architecture News* 40, 3, 72–83.

Ulrich Meyer and Peter Sanders. 1998. Delta-stepping: A parallel single source shortest path algorithm. In *Proceedings of the European Symposium on Algorithms (ESA'98)*. 393–404.

Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013a. Data-driven versus topology-driven irregular computations on GPUs. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. 463–474.

Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013b. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 147–156.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2, 40–53.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY,12–25.

Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. 2011. EigenCFA: Accelerating flow analysis with GPUs. *ACM SIGPLAN Notices* 46, 1, 511–522.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: A system for synthesizing concurrent graph programs. *ACM SIGPLAN Notices* 47, 10, 375–394.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 519–530.

G. Ramalingam and T. Reps. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 233–277.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 472–488.

Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. 2012. Paragon: Collaborative speculative loop execution on GPU and CPU. In *Proceedings of the Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, 64–73.

Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the Workshop on General Purpose Processing Using Graphics Processing Units (GPGPU-6)*. ACM, New York, NY, 76–85.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. *ACM SIGPLAN Notices* 48, 8, 135–146.

Richard M. Stallman and the GCC Developer Community. 2011. Using the GNU Compiler Collection. Retrieved November 18, 2015, from https://gcc.gnu.org/onlinedocs/gcc.pdf.

Morten Stockel and Soren Bog. 2008. *Concurrent Datastructures*. Technical Report IMM-BSC-2008-12. Technical University of Denmark.

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. 330–341.

Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. 2011. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 189–200.

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8, 103–111.

Shucai Xiao and Wu Chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDP'10)*. 1–12.

Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, 183–193.

Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6, 1543–1552.

Received June 2015; revised October 2015; accepted October 2015