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

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.

1. INTRODUCTION

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

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

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

It would be really helpful if a programmer can specify a graph algorithm in a hardware-independent manner and focus solely on the algorithmic logic. Unfortunately, such an approach – which essentially auto-parallelizes a sequential piece of code – provides limited performance in general when compared with a manually parallelized hardware-centric code by an expert.
Our goal in this work is to bridge this performance gap between an auto-generated code and a manually crafted implementation. We wish to let the programmer write the algorithm at a higher level (much higher than CUDA and OpenCL), without any hardware-centric constructs. To achieve performance close to that of a 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, while the second one allows our code generator to emit hardware-specific code.

Our specific contributions are given below.

— The design of Falcon, a DSL for general graph algorithms. Unlike previously reported languages, Falcon supports morph algorithms, that is, algorithms wherein the graph structure may also change, apart from the values at the nodes and the edges.
— Falcon’s code generation scheme for multicore CPU, single GPU, multi-GPU, and heterogeneous backends. Our compiler supports worklist based implementations of morph and local computation algorithms on CPU, that run much faster than most hand-written implementations.
— Falcon’s support for graph partitioning and execution of a single algorithm on the partitioned graph on the CPU and multiple-GPUs (for vertex-centric algorithms only).
— Performance analysis of Falcon: We generate CUDA and OpenMP code for morph algorithms such as Delaunay mesh refinement (DMR), survey propagation (SP) and dynamic single source shortest path (SSSP), as well as CUDA and OpenMP code for local computation algorithms. Performance of these and several other benchmarks are compared against the state-of-the-art DSL and framework-based implementations.

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

2. BENEFITS OF FALCON

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

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

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

3. RELATED WORK

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

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

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

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

Parallel Boost Graph Library [Gregor and Lumsdaine 2005] is a distributed version of BGL and SNAP [Bader and Madduri 2008] [Bader and Madduri 2005] is a stand-alone parallel graph analysis package. Ligra [Shun and Blelloch 2013] is a framework for writing graph traversal algorithms for multicore shared memory systems which used two different routines to map vertices and edges. Polymer [Zhang et al. 2015] is NUMA aware graph framework for multicore CPUs and it is built with a hierarchical barrier to get more parallelism and locality. Halide [Ragan-Kelley et al. 2013] is DSL targeting multicore-CPU and GPU for image processing applications. X-Stream [Roy et al. 2013] uses edge centric processing for graph applications rather than using vertex centric processing for algorithms such as SSSP and Strongly Connected Components. CuSha [Khorasani et al. 2014] proposes two new ways of storing graphs on GPU that has improved regular memory access patterns. Efficient implementations of local computation algorithms such as Breadth First Search (BFS) and Single Source Shortest Path (SSSP) on GPU have been reported several years ago [Harish and Narayanan 2007] [Harish et al. 2009]. There have also been successful implementations of other local computation algorithms such as n-body simulation [Bartscher and Pingali 2011], betweenness centrality [Sariyüce et al. 2013] and data flow analysis [Mendez-Lojo et al. 2012; Prabhu et al. 2011] on GPU. [Davidson et al. 2014] proposes different ways of writing SSSP programs on GPU along with their merits and demerits. It concludes that worklist-based implementation would not benefit much on GPU compared to that on a CPU.

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

4. OVERVIEW OF FALCON

4.1. Introduction

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

4.2. Example: Shortest Path Computation

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

**Falcon** needs each variable which resides on the GPU to have the <GPU> tag preceding the variable name in the declaration statement (Lines 1, 19). Being a graph-DSL, the type **Graph** is directly available in the language.

```c
1 int <GPU> changed = 0; // Variable on GPU
2
3 relaxgraph(Point <GPU> p, Graph <GPU> graph) {
4     p.uptd = false;
5     foreach (t in p.outnbrs) {
6         MIN(t.dist, p.dist + graph.getWeight(p, t), changed);
7     }
8 }
9
10 reset(Point <GPU> t, Graph <GPU> graph) {
11     t.dist = t.olddist = 1234567890; t.uptd = false;
12 }
13
14 reset1(Point <GPU> t, Graph <GPU> graph) {
15     if (t.dist < t.olddist) t.uptd = true;
16     t.olddist = t.dist;
17 }
18
19 //main function on rhs
20
21 //main function on rhs
22
23 main(int argc, char *argv[]) {
24     Graph hgraph; // graph on CPU
25     hgraph.addPointProperty(dist, int);
26     hgraph.getType() <GPU> graph; // graph on GPU
27     graph.addPointProperty(uptd, bool);
28     graph.addPointProperty(olddist, int);
29     hgraph.read(argv[1]) // read graph on CPU
30     graph = hgraph; // copy graph to GPU
31     foreach (t in graph.points) reset(t,graph);
32     graph.points[0].dist = 0; // source has dist 0
33     foreach (t in graph.points) reset1(t,graph);
34     while(1) {
35         changed = 0; //keep relaxing on GPU
36         foreach (t in graph.points) (t.uptd)
37             relaxgraph(t,graph);
38         if (changed == 0) break; //reached fix point
39         foreach (t in graph.points)
40             reset1(t,graph);
41         graph.dist = graph.dist; // copy all points
dist to CPU
42         for (int i = 0; i < hgraph.npoints; ++i)
43             printf("i=%d dist=%d\n", i,
44                 hgraph.points[i].dist);
45     }
46 }
```

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

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

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

The **relaxgraph()** function is called repeatedly (Line 29) and it keeps on reducing **dist** value of each **Point** (Line 5). The foreach in **relaxgraph()** is augmented with a condition (t.uptd)
that makes sure that only those points which satisfy the condition will execute the code inside the relaxgraph() function. In the first invocation of relaxgraph(), only the source node will perform the computation. Since multiple threads may update the distance of the same node (e.g., when relaxing edges \((u_1, v)\) and \((u_2, v)\)), some synchronization is required across the threads. This is achieved by providing atomic variants for commonly used operations. The MIN() function used by relaxgraph() is an atomic function that reduces dist atomically (if necessary) and if it does change, the third argument value will be set to 1 (Line 5). So, whenever there is a reduction in the value of dist for even one Point, the variable changed is set to 1. Line 3 makes updt property of each Point whose current value is true to false. After each call to relaxgraph(), the reset() function makes updt 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 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. The Lines 20 and 21 should be modified to add the properties to 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 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.

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

Table II. Data Types in Falcon

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

4.3.2. Graph. A Graph stores its points and edges in vectors points[] and edges[]. The method addEdgeProperty() 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 which are not necessarily direct functions of points and edges. For instance, such a function is used in Delaunay Mesh Refinement (DMR) [Chew 1993] code as the graph consists of a collection of triangles, each triangle with three Points and a few extra properties. The statement shown below illustrates the way DMR code uses this function for a Graph object, hgraph.

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

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

4.3.3. Set. A Set is an aggregate of unique elements (e.g., a set of threads, a set of nodes, etc.). A Set has a maximum size and it 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 MST algorithm [Chung and Condon 1996]. The way Set data type is declared in MST code is shown in Figure 2.

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

Figure 3 shows how minimum weight edges are marked in the MST computation. Function MinEdge(), which gets converted to a device function, takes three parameters: a Point to operate on, the underlying Graph object on the GPU, and a Set of points. Line 14 takes each outgoing neighbor of the Point p and checks whether those neighbors and p belong to the same set using the find() function. If not (Line 17), the code checks whether the edge \((p, t)\) has the minimum weight (Line 18). If it is indeed of minimum weight, the code tries to lock the Point using the single construct (see Section 4.5) in Line 19. If the locking is successful, this edge is added to the MST. After MinEdge() completes, each end-point of the edge which was newly added to the MST is put into the same Set using the union operation (performed in the caller).

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

Fig. 3: Finding the minimum weight edge in MST computation

Delaunay Mesh Refinement [Chew 1993] needs local Collection objects to store a cavity of bad triangles and to store newly added triangles. Hence, it can be implemented using dynamic arrays. Our implementation creates an initial array with a default size. When it gets full, it dynamically allocates another array of larger size, copies all the elements from the old array to the new array, and deallocates the old array. In general, repeated copying of elements is expensive. However, we significantly reduce this cost by repeated doubling of the array size. 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. Figure 4 shows how Collection objects are used in DMR code. Lines 2–3 declare a Collection object with the name pred on GPU and hpred on CPU which contains elements of type struct node.

Graph hgraph, <GPU>graph;
Collection pred[struct node (graph)];
Collection hpred[struct node (hgraph)];

Fig. 4: Declaring a Collection object on CPU and GPU in Falcon

struct node { //structure for triangle
  Point nodes[3], neighedgestart[3];
  struct_rec node neighbors[3];
  int isbad, isdel, obtuse, owner, dims, index;
};

Fig. 5: Recursive data structure in Falcon (used in DMR)

4.4. Variable Declaration

Variable declarations in Falcon can occur in two forms as shown with Point variables P0 and P1 below (Edge declarations are similar). Given a Graph object g, we say that g is the parent
of the points and edges in $g$.

Point $P_1$, $(\text{graph})P_0$; //parent Graph of $P_0$ is graph

When a point or edge variable has a parent Graph object, it can be assigned values from that parent only and whatever modifications we make to that object will be reflected in the parent Graph object. In the above example, $P_0$ can be assigned values that are Point objects of graph only (see also line 10 of Figure 3). But if a variable is declared without a parent and a value is assigned to it, it will be copied to a new location and any modification made to that object will not be reflected anywhere else (e.g., $P_1$ in the above example).

```c
int <GPU,1>changed; //allocate variable on GPU 1.
```

**Falcon** allows a programmer to specify on which GPU device the variable needs to be allocated with the optional integer argument along with <GPU> tag as shown above.

**Falcon** has a new keyword named struct_rec, that is used to declare recursive data structures. In C, a recursive data structure can be implemented using pointers and the malloc() library function. With struct_rec, a programmer can support a recursive data structure without explicitly using pointers, (like in Java). The triangle property declaration used in DMR is shown in Figure 5 (a triangle has at max three neighboring triangles).

### 4.5. Parallelization and Synchronization Constructs

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

#### 4.5.1. single statement

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

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

| 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 executes stmt block1 and others execute stmt block2. |

**Table III. Single statement in Falcon**

**Falcon** supports two variants for single, as given in Table III: with one item and with a Collection of items. In both the variants the else block is optional (Figure 3, Line 19). The first variant tries locking one item. As it is a non-blocking entry function, if multiple threads try to get a lock on the same object, only one will be successful, others will fail. In the second variant, a thread tries to get a lock on a Collection of items given as an argument. This allows a programmer to implement cautious forms of algorithms wherein all the shared data (e.g., a set of neighboring nodes) are locked before proceeding with the computation. A thread succeeds if all the elements in the Collection object are locked by that thread. As an example, a thread in DMR code tries to get a lock on a cavity, which is a Collection of triangles. In both the variants, the thread that succeeds in acquiring a lock executes the code following it and if the optional else block is present, all the threads that do not acquire the lock execute the code inside the else block.
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 object

Table IV. foreach statement in Falcon

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

Table V. Iterators for foreach statement in Falcon

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

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 the points and edges in parallel. An iterator called pptyname is generated automatically when a new property is added to a Graph object using addProperty() function. This is often used in morph algorithms. When a property triangle is added to 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 operators such as ReduxSum, which sums a set of items and ReduxMul which multiplies a set of items are provided by Falcon. We leave the support for arbitrary associative functions as reduction operations as future work.

4.6. Library Functions

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

4.7. Graph Partitioning

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

5. CODE GENERATION

We now explain how the Falcon compiler generates code (code fragments are shown with macro statements to make the code readable, but these macros are not a part of the compiler generated code). Falcon extends the C language grammar to support additional constructs. The compiler generates CUDA/C++ code. Currently, it supports two types of graph representation: (i) Compressed Sparse Row (CSR) format, and (ii) Coordinate List (COO) or Edge List format. Graphs are stored as C++ classes in Falcon generated code. The GGraph and HGraph classes are used to store a graph.
on the GPU and the CPU respectively, and both inherit from a parent Graph class. The Graph class has an extra field (of type void *) which stores all the properties added to a Graph object using addPointProperty(), addEdgeProperty(), and addProperty(). The Point and Edge data types can have either integer (default) or floating point values and are stored in a union type with fields ipe and fpe respectively. The generated code is compiled with nvcc and g++. Figure 6 gives an overview of how parallelization and synchronization is done for CPU and GPU. The Falcon compiler names for all data types and functions specific to CPU and GPU start with H(Host) and G(Gpu) respectively in the generated code.

5.1. Type Checking

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

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

Fig. 7: Allocating extra property for Graph object on GPU

5.2. Properties

Point and Edge are converted to integer ids. All the extra properties of a Graph object are stored in the extra field, and can be type casted to any structure. By default, extra properties are stored in a structure with the name struct_objectname and are assigned to the extra field of a Graph object. If a Graph object is created by getType() function, its extra properties are assigned to a structure with the name struct_parentobjectname, which will have fields for extra properties of the parent object and all the objects created by the getType() compile time function. In the SSSP example (Figure 1), Graphs on GPU and CPU are both allocated in a structure with the same name as the GPU Graph object is being created with a call of getType(). Figure 7 shows how extra properties of the Graph object on the GPU in the SSSP computation are allocated. For the CPU Graph object (hgraph), only dist field is allocated using malloc(), as olddist and 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 (resp.). Each of these classes has the same functions named, union to union two sets and find to find the representative key for an element. By default, the key for a subset will be an integer number, which denotes the maximum value of an element in that subset.

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

5.4. Foreach Statement

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

The outermost loop is retained as a foreach statement and is converted to a CUDA kernel call / OpenMP pragma (except for Collection on CPU) in the generated code. Figure 8 shows the code generated for the relaxgraph() function and its foreach call from Figure 1, with the target being GPU. Since foreach statement inside relaxgraph() is nested inside foreach statement from main(), the foreach inside relaxgraph() is converted to a simple for loop. The variable TPB (Threads Per Block) corresponds to (MaxThreadsPerBlock - MaxThreadsPerBlock % CoresPerSM) for the GPU device on which the CUDA kernel is being called.

We also make sure that a kernel executes by splitting a kernel call into multiple calls, if the number of threads or blocks for the kernel call is above the allowed value for device. Each Edge in Falcon stores two values in the edges array, the destination Point and weight of the Edge. When a program uses innbrs iterator and outnbrs iterators, the inedges arrays of the Graph class stores two fields: source Point of the incoming Edge and an index in to the edges array which can be used to find out weight of the incoming Edge, which is stored in edges arrays.

Figure 9 shows the code generated for relaxgraph() function and its foreach statement when SSSP is written for a multicore CPU. The variable TOT_CPU stores the number of CPU cores available. The MIN function is converted to GMIN for GPU and HMIN for CPU. This convention is used throughout Falcon, as can be seen with Graph type converted to HGraph or GGraph based on where it is allocated.
Falcon stores the beginning index of neighbors of a Point in the index field of Graph class and total neighbors of point is found by taking difference of index value of nespoint and this point. The foreach statement in relaxgraph() processes all the neighbors of a Point serially, using a simple for loop. Similar code is generated for other iterators of Point and Edge data type.

We have experimented with warp-based code generation as well. However, we find that performance improvement is not always positive across benchmarks. Details of our code generation are provided in [Unnikrishnan et al. 2015]. Code in figure 10 shows the warp-based code generated for relaxgraph() function of SSSP example in Figure 1. Falcon compiler do program analysis to find out which properties of the thread are read only and allocates it in shared memory. For the relaxgraph() function dist field and index field of Graph class are readonly and allocated then in shared memory using the structure warprelaxgraph. It finds out size taken by the structure and allocates for each thread sufficient number of threads so that each thread can store required shared data in the arry of structures with name SM_THREAD. To process SIZE number of elements per thread(position 0 to SIZE-1), the nbbs field should have (SIZE+1) locations to store nbbs field so that total neighbors of element at location SIZE-1 can found as diferenc (nbbs[SIZE]-nbbs[SIZE-1]). Based on available shared memory for a GPU per SM(symmetric Multiprocessor) and size of the structure used to shared data, Falcon automatically finds out values for macro variables WORK_PER_THREAD,SHMEM_SIZE. The macro variable CORES_PER_SM is number of cores available in a SM of the GPU. Each thread copied values to shared memory then do the processing. The macro variables are added in header file of the generated code and kernel is called with CORES_PER_SM threads per block.

5.5. parallel sections, Multiple GPUs and Multiple Graphs

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

For parallel kernel execution on different GPUs, each foreach statement should be placed inside a different section of the parallel sections statement. The parallel sections statement gets converted to a OpenMP parallel region pragma, which makes it possible for the code segments in different sections inside the parallel sections to run in parallel. A programmer can disable the multi-GPU support using a compile time option. 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 11 shows how SSSP and BFS are computed at the same time on different GPUs using a parallel sections statement of Falcon. An Important point to be noted here relates to how changed variable is used in the code. if we declare changed as shown in Line 1 of Figure 11, it will be allocated in GPU device 0. So, to ensure that changed appears in each device, it is added as a Graph Property in Line 5.
#define t (((struct struct_hgraph *)(graph.extra)))

struct warprelaxgraph{
  int dist;
  int nbrs;
};

__global__ void relaxgraph ( GGraph graph ) {
  int id2=( blockIdx.x * blockDim.x + threadIdx.x)*WORK_PER_THREAD;
  __shared__ struct warprelaxgraph SM_THREAD[SHMEM_SIZE];
  int i;
  int NUM_WARPS=graph.npoints/WARP_SIZE;
  int SIZE=WORK_PER_THREAD+1;
  int OFF=threadIdx.x%CORES_PER_SM;
  OFF=OFF*SIZE;
  int OFF1;
  OFF1=OFF-threadIdx.x%CORES_PER_SM;
  int x1=0;
  if(id2>=graph.npoints)return;
  for (int id = OFF1; id <OFF1+SIZE; id++) {
    if(((id2+x1) < graph.npoints) && t->updated[id2+x1]==true)) {
      t->updated[id2+x1]=false;
      SM_THREAD[id].dist=t->dist[id2+x1];
      SM_THREAD[id].nbrs=graph.index[id2+x1];
      SM_THREAD[id+1].nbrs=graph.index[id2+x1+1];
      int falcft0=SM_THREAD[id+1].nbrs-SM_THREAD[id].nbrs;//used shared memory,total out going edges
      int falcft1=SM_THREAD[id].nbrs;//starting index of neighbors
      for (int falcft2 = 0; falcft2 < falcft0; falcft2++) {
        int ut0=2*(falcft1+falcft2);
        int ut1=graph.edges[ut0].ipe;
        int ut2=graph.edges[ut0+1].ipe;
        GMIN(&(t->dist[ut1]),SM_THREAD[id].dist+ut2,changed);//MIN with shared memory part of second argument.
      }
    }
    x1++;
  }
}

Fig. 10: Warp based code generated for relaxgraph() function SSSP, Figure 1

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

Fig. 11: Multi-GPU BFS and SSSP in Falcon.
5.6. Inter-Device Communication

Copying data between the CPU and the GPU is translated to cudaMemcpy which has different forms for the various assignment statements in Falcon. When an entire property of Graph, say Point or Edge is copied from GPU or to GPU, a cudaMemcpy operation is called to transfer a block of data. Falcon allows direct usage of GPU variables of basic types such as int, bool etc. inside CPU code. Line 30 and Line 28 of the SSSP example in Figure 1 are converted to CUDA code as shown in Figures 13 and 14 (resp).

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

\[ hgraph.dist = graph.dist; \] // Line 33 of Figure 1.

The generated CUDA code for this statement is shown in Figure 12. The above statement needs two cudaMemcpy operations as graph.extra is a GPU location and we cannot access graph.extra.dist in cudaMemcpy, as this implies dereferencing a device location (something that cannot be done from the host). A programmer can use The GPU Graph object directly in the printf statement and Falcon compiler generates code to copy dist value of all points to temporary pointer variable and use that in printf statement.

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

5.7. Synchronization statement

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

There is no support for a grid barrier in CUDA and we have implemented it as given in [Xiao and Chun Feng 2010]. The CPU code uses barrier provided by OpenMP. The way single statement is
used in DMR is shown in Figure 28. Here \textit{pred} is a \texttt{Collection} object which stores the set of all \textit{triangles} in the cavity. If a lock is obtained on all the \textit{triangles} then the cavity is updated else the corresponding thread is aborted.

Pseudo Code in Lines 5-9 Figure 28 get converted to the CUDA code shown in Figure 29. Both \texttt{GPU} and \texttt{CPU} versions follow the above code pattern, with appropriate \texttt{GPU} and \texttt{CPU} functions. We lock the \textit{triangles} based on the thread id and if two or more cavities overlap only the thread with the lowest thread id will succeed in locking the cavity and others abort. The global barrier makes sure that the operations of all threads are complete up to the barrier before any thread can proceed. This generated code is similar to that used in LonestarGPU.

---

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

Fig. 15: Usage of single statement in DMR(Pseudo code)

```c
#define t ((struct struct_graph *)(graph.extra))
for(int i=0;i<pred.size;i++)
  t->owner[pred.D_Vec[i]] = id;
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
  }

  gpu_barrier(++goal, arrayin, arrayout); // global barrier
  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. 16: Generated CUDA code

The first variant of single statement in Table III that locks a single object does not need a barrier. It uses the compare and swap variant of CUDA [Nickolls et al. 2008] and GCC [Stallman et al. 2011] for \texttt{GPU} and \texttt{CPU} respectively. A barrier for the entire function will increase the running time and so we adopted a barrier free implementation of a single statement for its first form.

This type of single statement is normally used in local computation algorithms such as MST computation. In order for the single to work properly, the property value must be reset to zero before entering the function in which single is executed.

5.8. Reduction Function

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

5.9. Modifying Graph Structure

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

For adding a Point or an Edge we rely on atomics. For a Graph object with the name say graph, we add global variables \texttt{falcgraphpoint, falcgraphedge} which will be initialized to the number of points and edges in graph(resp.). When we call \texttt{graph.addPoint} in the Falcon program, that code will be replaced by a call to an automatically generated function \texttt{falcaddgraphpointfun()}. This function atomically increments \texttt{falcgraphpoint} by one. Analogous functions exist for Edge and properties added using the \texttt{addProperty} function. Currently, none of properties (attributes)
associated with graph elements can be auto-deleted (including the one added using addProperty); their deletion must be explicitly coded by the programmer. DMR deletes triangles 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 pre-allocated memory may not be sufficient. A call to compiler generated realloc() function is inserted automatically after the code that modifies the Graph size. This realloc() function considers current size, the change in size and the available extra memory allocated and performs Graph reallocation, if necessary.

While it is true that graph algorithms exhibit irregularity, overall, the following help us achieve better coalescing and locality (i) CSR representation enables accessing nodes array in a coalesced fashion. It also helps achieve better locality as edges of a node are stored contiguously. (ii) Shared memory accesses for warp-based execution and reductions help improve memory latency. (iii) Optimized algorithms. Note that a high-level DSL allows us to tune an algorithm easily, such as the SSSP optimization discussed in Section 4.

5.10. Heterogeneous Execution in Falcon using Graph Partitioning

When a Graph object does not fit into GPU memory, the programmer can make use of the graph partitioning functions available in Falcon. Falcon currently supports partitioned execution with one CPU and multiple GPUs. The partitioning algorithm, communication mechanism, and subgraph storage structures used in Falcon have been derived from Totem [Gharaibeh et al. 2013][Gharaibeh et al. 2012]. But Falcon hides all the internal details from the programmer. Falcon supports random partitioning and 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 (same is the case with Totem). A non-vertex-centric algorithm requires edge-based processing and this may result in huge amounts of communication, as the number of edges in a graph is usually much higher than the number of nodes. This will be explored in future work.

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

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

Falcon code in Figure 17 shows how SSSP computation can be performed on an input using both GPU and CPU. The makePartition function in Line 14 of Figure 17 partitions the graph into two parts, one each on CPU (argument 1) and GPU (argument 2) using the partition algorithm based on the degree of nodes in a graph (argument 3).

After a computation step, the current values of remote nodes are communicated to the partition in which the remote node is actually present. The updating function, updatePartition() (Line 22) applies the function fun1 (defined in Line 1 and specified as shown in Line 15) to update the...
value. The update function does not need atomic operations as each thread is accessing a different
description. The Falcon compiler optimizes data transfers between partitions by sending the values
of only required properties to remote partition (e.g., property values of Point incom, which are
read in fun1, in Figure 17).

```c
fun(Point ori, Point incom){
    if(ori.dist > incom.dist)
        orig.dist = incom.dist
}
```

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

```c
main(int argc,char *argv[]){
    HGraph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.addProperty(changed,int);
    hgraph.read(argv[1]);
    hgraph.makePartition(1,1,SORT_BY_DEGREE);
    hgraph.updateFunction(fun1);
    foreach(t In hgraph.points)t.dist=1234567890;
    hgraph.updatePartition();
}
```

Fig. 17: Partitioned SSSP algorithm (unoptimized)

For partitions in 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
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 the CPU,
respectively. The if statement checks whether the value in the variable changed is unchanged (in
both the partitions). If a programmer wants to execute only on multiple-GPUs or multiple-GPUs
and CPU, the first two arguments are required to be modified. A programmer can also specify the
percentage of a Graph object to be allocated on the CPU and GPUs using command line arguments.

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

6. EXPERIMENTAL EVALUATION

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

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

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

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Falcon GPU</th>
<th>Green-Marl</th>
<th>Galois</th>
<th>Totem GPU</th>
<th>Falcon GPU</th>
<th>Lonestar GPU</th>
<th>Totem GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>26</td>
<td>24</td>
<td>310</td>
<td>600</td>
<td>28</td>
<td>140</td>
<td>200</td>
</tr>
<tr>
<td>SSSP</td>
<td>25</td>
<td>310</td>
<td>60</td>
<td>38</td>
<td>170</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>MST</td>
<td>113</td>
<td>N.A.</td>
<td>1011</td>
<td>N.A.</td>
<td>103</td>
<td>420</td>
<td></td>
</tr>
<tr>
<td>DMR</td>
<td>302</td>
<td>N.A.</td>
<td>308</td>
<td>860</td>
<td>N.A.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>198</td>
<td>N.A.</td>
<td>401</td>
<td>N.A.</td>
<td>185</td>
<td>420</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>51</td>
<td>N.A.</td>
<td>56</td>
<td>165</td>
<td>N.A.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.1. Local Computation Algorithms

Figure 18 shows the results for BFS, SSSP and MST on GPU and Figure 19 shows the results for BFS and SSSP on CPU. MST speedup on CPU is shown in Figure 20. We experimented with...
several graph types (such as the Erdős-Rényi model random graphs [Erdős and Rényi 1960], road networks, and scale-free graphs) and have shown results for two representative graphs from each category, with several million edges. Details can be seen in Table VI. Road network graphs are real road networks of USA [DIMACS 2009], have less variance in degree distribution, but have large diameter. Scale-free graphs have been generated using GTGraph [Bader and Madduri 2006] tool, have a large variance in degree distribution but exhibit small-world property. Random graphs have been generated using the graph generation tool available in Galois.

SSSP. Results for SSSP on GPU have been plotted as speedup over best time reported by LonestarGPU variants (worklist based SSSP and Bellman-Ford style SSSP). We find that Falcon SSSP (Figure 1) is faster than LonestarGPU. This is due to the optimization used in the Falcon program using the uptd field, which eliminates many unwanted computations. For rmat2 input worklist based SSSP of LonestarGPU went out of memory and speedup shown is over slower Bellman-Ford variant of the SSSP code (Figure 1) on CPU with 12 threads is about 8× slower than that of the same on GPU. It is the worklist based Δ-stepping algorithm which made CPU code fast. BFS and MST also benefit considerably from worklist based execution on GPU.

BFS. Results for BFS on GPU are compared as speedup over the best running times reported by LonestarGPU. We took the best running times reported by worklist based BFS and Bellman-Ford variant BFS implementations. The worklist based BFS performed faster only for road network inputs. 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. GreenMarl failed to run on rmat input giving a runtime error on std::vector::reverse(). It is important to note that Bellman-Ford variant of the SSSP code (Figure 1) on CPU with 12 threads is about 8× slower than that of the same on GPU. It is the worklist based Δ-stepping algorithm which made CPU code fast. BFS and MST always outperformed Galois BFS, due to our optimizations (Figure 19(b)). Totem and GreenMarl are again slower on road inputs. Totem performed better than Falcon BFS on GPU for scale free graphs. Totem runs algorithms using graph partitioning which benefits graphs that follow the power law distribution, and rmat graphs do follow the power law [Gharaibeh et al. 2012]. The speedup for BFS on GPU is shown for Totem and Falcon with respect to LonestarGPU in Figure 18(b).

MST. LonestarGPU has a Union-Find based MST implementation. Falcon GPU code for MST always outperformed that of LonestarGPU for all inputs, with the help of better implementation of Union-Find that Falcon has for GPU. But our CPU code showed a slowdown compared to Galois (about 2× slowdown). Galois has a better Union-Find implementation based on object location as key. The Speedup for MST on GPU is shown in Figure 18(c) and same for CPU is shown in Figure 20.

Multi-GPU. Figure 21 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 21 and values in Figure 18, 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 GPU and CPU in parallel using the parallel sections statement. A Programmer can decide where to run a program by allocating a Graph object on GPU or CPU, which can be specified in a declaration statement with or without using <GPU> tag. He/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 CPU (because it is
slow on GPU) and for random and rmat graph inputs, on GPU. The effort required to modify codes for CPU or GPU is minimal with Falcon.

We have Falcon implementations of many other graph algorithms such as page ranking, betweenness centrality, etc., and these can be found in [Unnikrishnan et al. 2015]. We found it easy to implement such algorithms in Falcon without worrying about the details of the underlying architecture.

<table>
<thead>
<tr>
<th>Input</th>
<th>Copy Graph to GPU</th>
<th>From initialization to end of while</th>
<th>While loop</th>
<th>CopyResult to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Falcon GPU</td>
<td>Lonestar GPU</td>
<td>Totem GPU</td>
<td>Falcon GPU</td>
</tr>
<tr>
<td>rand1</td>
<td>192.1</td>
<td>210.7</td>
<td>418.7</td>
<td>494.7</td>
</tr>
<tr>
<td>rand2</td>
<td>460.3</td>
<td>541.7</td>
<td>848.2</td>
<td>845.7</td>
</tr>
<tr>
<td>rmat1</td>
<td>246.77</td>
<td>252.5</td>
<td>250.1</td>
<td>691</td>
</tr>
<tr>
<td>rmat2</td>
<td>489.5</td>
<td>508.8</td>
<td>490.7</td>
<td>1558.9</td>
</tr>
<tr>
<td>road1</td>
<td>158.7</td>
<td>225.9</td>
<td>464</td>
<td>33806.3</td>
</tr>
<tr>
<td>road2</td>
<td>192.2</td>
<td>324.3</td>
<td>7596.6</td>
<td>78127.3</td>
</tr>
</tbody>
</table>

Table VIII. Running Time(in Ms) SSSP on GPU(Falcon-GPU,LonestarGPU,Totem-GPU)

<table>
<thead>
<tr>
<th>Input</th>
<th>Copy Graph to GPU</th>
<th>From initialization to end of while</th>
<th>While loop</th>
<th>CopyResult to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Falcon GPU</td>
<td>Lonestar GPU</td>
<td>Totem GPU</td>
<td>Falcon GPU</td>
</tr>
<tr>
<td>rand1</td>
<td>192.1</td>
<td>495.3</td>
<td>418</td>
<td>100.84</td>
</tr>
<tr>
<td>rand2</td>
<td>432</td>
<td>664</td>
<td>819</td>
<td>199</td>
</tr>
<tr>
<td>rmat1</td>
<td>279.1</td>
<td>568.2</td>
<td>251.4</td>
<td>158.6</td>
</tr>
<tr>
<td>rmat2</td>
<td>474.7</td>
<td>704.7</td>
<td>498.7</td>
<td>319.2</td>
</tr>
<tr>
<td>road1</td>
<td>151.7</td>
<td>379.7</td>
<td>462.7</td>
<td>380.1</td>
</tr>
<tr>
<td>road2</td>
<td>424.4</td>
<td>785.3</td>
<td>597.3</td>
<td>604.1</td>
</tr>
</tbody>
</table>

Table IX. Running Time(in Ms) BFS on GPU(Falcon-GPU,LonestarGPU,Totem-GPU)

<table>
<thead>
<tr>
<th>Input</th>
<th>Copy Graph to GPU</th>
<th>From initialization to end of while</th>
<th>While loop</th>
<th>CopyResult to GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Falcon GPU</td>
<td>Lonestar GPU</td>
<td>Totem GPU</td>
<td>Falcon GPU</td>
</tr>
<tr>
<td>rand1</td>
<td>632.5</td>
<td>837.7</td>
<td>1521.3</td>
<td>3763.7</td>
</tr>
<tr>
<td>rand2</td>
<td>1298.7</td>
<td>1217</td>
<td>4511</td>
<td>9379.2</td>
</tr>
<tr>
<td>rmat1</td>
<td>663</td>
<td>653.1</td>
<td>3641.1</td>
<td>3782.4</td>
</tr>
<tr>
<td>rmat2</td>
<td>1262.2</td>
<td>1241.1</td>
<td>7889.7</td>
<td>7907</td>
</tr>
<tr>
<td>road1</td>
<td>566.9</td>
<td>189.3</td>
<td>881.4</td>
<td>3146.4</td>
</tr>
<tr>
<td>road2</td>
<td>923.6</td>
<td>743.7</td>
<td>1072.67</td>
<td>4333.4</td>
</tr>
</tbody>
</table>

Table X. Running Time(in Ms) MST on GPU(Falcon-GPU,LonestarGPU)
### Table XI. Running Time(in Ms) for SSSP on CPU

<table>
<thead>
<tr>
<th>Input</th>
<th>Galois-1</th>
<th>Galois-12</th>
<th>Falcon-CPU</th>
<th>Totem-CPU</th>
<th>Green-Marl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand1</td>
<td>5832.5</td>
<td>680.5</td>
<td>420.5</td>
<td>716</td>
<td>1017</td>
</tr>
<tr>
<td>rand2</td>
<td>12074.5</td>
<td>1464.3</td>
<td>949.3</td>
<td>1468.1</td>
<td>1952</td>
</tr>
<tr>
<td>rmat1</td>
<td>3345</td>
<td>501.4</td>
<td>397.6</td>
<td>512.1</td>
<td>killed</td>
</tr>
<tr>
<td>rmat2</td>
<td>7133.5</td>
<td>995</td>
<td>825.9</td>
<td>1009.1</td>
<td>killed</td>
</tr>
<tr>
<td>road1</td>
<td>2042</td>
<td>320</td>
<td>325.5</td>
<td>2387.7</td>
<td>1225.2</td>
</tr>
<tr>
<td>road2</td>
<td>3063.5</td>
<td>520</td>
<td>482.7</td>
<td>4686.2</td>
<td>6127</td>
</tr>
</tbody>
</table>

### Table XII. Running Time(in Ms) for BFS on CPU

<table>
<thead>
<tr>
<th>Input</th>
<th>Galois-1</th>
<th>Galois-12</th>
<th>Falcon-CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand1</td>
<td>49861.8</td>
<td>5036</td>
<td>9795</td>
</tr>
<tr>
<td>rand2</td>
<td>110804</td>
<td>11419</td>
<td>20827</td>
</tr>
<tr>
<td>rmat1</td>
<td>29188</td>
<td>6910</td>
<td>14503</td>
</tr>
<tr>
<td>rmat2</td>
<td>63372</td>
<td>14972</td>
<td>36784</td>
</tr>
<tr>
<td>road1</td>
<td>11180</td>
<td>1402</td>
<td>1421</td>
</tr>
<tr>
<td>road2</td>
<td>12013</td>
<td>1510</td>
<td>1615</td>
</tr>
</tbody>
</table>

### Table XIII. Running Time(in Ms) for MST on CPU

6.2. Morph Algorithms

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

**Delaunay Mesh Refinement (DMR).** DMR implementation in LonestarGPU relies on a global barrier, which can be implemented either by returning to the CPU and launching another kernel, or by emulating a grid-barrier in software [Xiao and Chun Feng 2010]. LonestarGPU uses the latter approach as it allows saving the state of the computation in local and shared memory across barriers inside the kernel (which is infeasible in the first approach where the kernel is terminated) and this approach is used in Falcon DSL code as well. Unfortunately, grid-level barriers pose a limit on the number of threads with which a kernel can be launched, as all the thread-blocks need to be resident and all the threads must participate in the barrier; otherwise, the kernel execution hangs. Therefore, both LonestarGPU and Falcon-generated code restrict the number of launched threads, thereby limiting parallelism. However, it avoids costly global memory access. This is also observable in other morph algorithm implementations needing a grid-barrier. Figure 22(a) and 22(b) show the performance comparison of DMR code for GPU and CPU on input meshes containing a large number of triangles in the range 0.5 to 10 million. Close to 50% of the triangles in each mesh are initially bad (that is, they need to be processed for refinement). Galois goes out of memory for 10 million triangles or more, and terminates. Falcon code is about 10% slower compared to LonestarGPU.
code and both used the same algorithm. This can be due to the inefficiency arising from conversion of DSL code to CUDA code. Speedup shown is for mesh refinement code (including communication involved during that time), after reading mesh.

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

The first four rows of Table XIV 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 with LonestarGPU-1.0 and Galois-2.1 codes. New versions of both these frameworks use a new algorithm, which is yet to be coded in Falcon. Multicore Galois goes out-of-memory for higher values of (K, N, M), whereas LonestarGPU and Falcon versions complete successfully.
LonestarGPU allocates each property of clause and literal in separate arrays whereas in Falcon, each property of clause and literal is put in structures, one each for clause and literal. Galois has a worklist based implementation of the algorithm. Also both Galois and LonestarGPU work by adding edges from clauses (Point in Graph) to each literal (Point in Graph) in the clause. But Falcon takes a clause as an extra property of the Graph (like triangle was used in DMR) and that property stores literals (Points) of the clause in it. So our Graph does not have any explicit edges, and literals of a clause (which correspond to edges) can be accessed very efficiently from the clause property of the Graph. We find that Falcon code runs faster than that of both Galois and LonestarGPU. Writing an algorithm that maintains a clause as a property of a Graph in LonestarGPU and Galois is not an easy task.

**Dynamic SSSP.** In a dynamic Single Source Shortest Path (SSSP) algorithm, edges can be added or deleted dynamically. A dynamic algorithm where only edges get added (deleted) is called as an incremental (decremental) algorithm, whereas algorithms where both insertion and deletion of edges happen are called fully dynamic algorithms [Frigioni et al. 1998]. We have implemented an incremental dynamic algorithm on GPU and CPU using Falcon. We have used a variant of the algorithm by [Ramalingam and Reps 1996]. Insertions are carried out in chunks and then SSSP is (incrementally) recomputed. We found it difficult to add dynamic SSSP to the Galois system, because no Graph structure that allows efficient addition of big chunk of edges to an existing Graph object was found. LonestarGPU code has been modified to implement dynamic SSSP, and we compare it with our CPU and GPU versions. Falcon looks at functions used in programs that modify a Graph structure (addPoint(), addEdge(), etc.) and converts a Graph read() function in Falcon to the appropriate read() function of the HGraph class. For dynamic SSSP, the read() function allocates...
Fig. 23: Heterogeneous Execution speedup Comparison over single threaded CPU (SSSP,BFS).

6.3. Heterogeneous Execution with Graph Partitioning

Falcon supports execution of vertex centric algorithms on CPU and multiple-GPUs using graph partitioning. We have collected results for two random graphs and three RMAT graphs. Random graphs are with 64M nodes(rand64) and 128M nodes(rand128) with number of total edges being four times the number of nodes. RMAT graphs are with 50M nodes(rmat50), 60M nodes(rmat60) and 80M nodes(rmat80) with total number of edges being ten times the number of nodes. Results are shown for SSSP and BFS on these inputs for execution on two GPUs (Figure 23(a)), and two GPUs and one CPU (Figure 23(b)) as compared to execution over single threaded CPU code. The reader should note that partitioned execution is to be used only when the graph does not fit into single GPU or single (multi-core) CPU memory. We utilized the GPU memory to the maximum possible extent for these large graphs. The rand128 input and rmat80 inputs did not fit in two GPUs and hence is executed on two GPUs and one CPU. The Totem framework and Falcon code were run on multi-GPU by enabling peeraccess and this is faster than code using UVA. The peeraccess method needs GPUs to be on same I/O Hub and so we used two GPUs(Fermi C2075 and Fermi C2050) which are on the same I/O Hub in our multi-GPU machine. Totem needed recompilation for compute capability 2.0 and modification of code to assign GPU partitions to use devices with peeraccess. Our results were collected with ordered partitioning (because it worked 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. Absoulute running time for two GPUs execution can be found in Table XV and for execution on two GPUs and one CPU in Table XVI. Experiments with two GPUs where run with equal distribution of vertices and its edges between GPUs. We found it is beneficial to execute rand64 and rmat50 with 90% graph on GPU (compared to two GPU execution) and rest on CPU and the running time was better than CPU alone execution. But when % of graph on GPU becomes around 50% or less the performance decreases due to communication overhead. For execution with two GPUs and one GPU, for rand128 input, 60% of graph object was on CPU while for rmat80 40% of graph object was on CPU and in both the cases remaining part of graph was equally distributed on two GPUs.
Table XV. Running Time (in Seconds) Partitioned execution on 2 GPUs

<table>
<thead>
<tr>
<th>Input</th>
<th>Galois-1</th>
<th>Falcon-MultiGPU</th>
<th>Totem-MultiGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand64-sssp</td>
<td>84.93</td>
<td>10.31</td>
<td>11.28</td>
</tr>
<tr>
<td>rmat50-sssp</td>
<td>76.48</td>
<td>8.46</td>
<td>9.05</td>
</tr>
<tr>
<td>rmat60-sssp</td>
<td>164.98</td>
<td>16.20</td>
<td>20.74</td>
</tr>
<tr>
<td>rand64-bfs</td>
<td>24.26</td>
<td>2.81</td>
<td>2.53</td>
</tr>
<tr>
<td>rmat50-bfs</td>
<td>16.83</td>
<td>2.36</td>
<td>1.93</td>
</tr>
<tr>
<td>rmat60-bfs</td>
<td>26.92</td>
<td>3.40</td>
<td>2.58</td>
</tr>
</tbody>
</table>

Table XVI. Running Time (in Seconds) Partitioned execution on 2 GPUs and 1 CPU

<table>
<thead>
<tr>
<th>Input</th>
<th>Galois-1</th>
<th>Falcon-MultiGPU</th>
<th>Totem-MultiGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand128-sssp</td>
<td>302.71</td>
<td>50.21</td>
<td>46.91</td>
</tr>
<tr>
<td>rmat80-sssp</td>
<td>257.78</td>
<td>44.46</td>
<td>46.71</td>
</tr>
<tr>
<td>rand128-bfs</td>
<td>38.38</td>
<td>9.25</td>
<td>8.99</td>
</tr>
<tr>
<td>rmat80-bfs</td>
<td>40.96</td>
<td>8.48</td>
<td>8.26</td>
</tr>
</tbody>
</table>

7. CONCLUSION AND FUTURE WORK

We presented Falcon, a domain specific language for expressing graph algorithms. It supports writing explicitly parallel programs, thus retaining efficiency. By enabling an algorithmic specification at a higher level, it allows easy changes to the code and also its maintenance. Salient features of Falcon are that it supports morph algorithms, wherein the underlying graph structure may change and provides support for heterogeneous architecture, multi-GPU systems and multi-core CPUs. We illustrated its expressibility by generating CUDA and OpenMP code for morph algorithms such as Delaunay mesh refinement, survey propagation and dynamic SSSP. We showed that writing code for CPU and GPU are similar, except in the case where variables in GPU need to be annotated with 

<GPU> tag and we showed that the generated code performs close to (and sometimes better than) their hand-tuned implementations. We also presented preliminary results of execution of vertex-centric algorithms on partitioned graphs. In the future, the portability of Falcon will be improved by supporting OpenCL as the backend and by extending Falcon support for CPU Clusters. Automatic code generation without the programmer explicitly specifying the location of Graph objects and supporting speculation with rollback are also on the cards.

REFERENCES


A:28


Mark Harris. 2007. Optimizing Parallel Reduction in CUDA.


Shucai Xiao and Wu chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. (April 2010).
8. APPENDIX
//SSSP DSL CODE
int <GPU> changed = 0;
relaxgraph(Point <GPU> p, Graph <GPU> graph) {
    p.uptd=false;
    foreach( t In p.outnbrs ){
        MIN(t.dist, p.dist + graph.getWeight(p, t), changed);
    }
}
reset( Point <GPU> t, Graph <GPU> graph) {
    t.dist=1234567890;
    t.uptd=false;
    t.olddist=1234567890;
}
reset1( Point <GPU> t, Graph <GPU> graph) {
    if( t.dist < t.olddist) t.uptd=true;
    t.olddist=t.dist;
}
main(int argc, char *argv[]) {
    Graph hgraph; // graph on CPU
    hgraph.addPointProperty(dist, int);
    hgraph.getType() <GPU> graph; // graph on GPU
    graph.addPointProperty(uptd, bool);
    graph.addPointProperty(olddist, int);
    hgraph.read(argv[1]) // read graph on CPU
    graph = hgraph; // copy graph to GPU
    foreach (t In graph.points)reset(t,graph); // initialize GPU graph object properties
    graph.points[0].dist = 0;
    graph.points[0].uptd=true;
    while(1) {
        changed = 0; //keep relaxing on GPU
        foreach(t In graph.points) (t.uptd) relaxgraph(t,graph);
        if(changed == 0)break;//reached fix point
        foreach(t In graph.points)reset1(t,graph)
    }
    for(int i = 0; i < graph.npoints; ++i)
        printf("i=%d dist=%d\n", i, graph.points[i].dist);
}
//HEADER FILE FOR SSSP EXAMPLE GENERATED CODE
#include "HGraph.h"
#include "HSet.h"
#include "thrust.cu"
#include <sys/time.h>
#include </usr/local/cuda/include/cuda.h>
#include </usr/local/cuda/include/cuda_runtime_api.h>
#include <unistd.h>
#include </usr/local/cuda-7.0/samples/0_Simple/simplePrintf/cuPrintf.cu>

struct struct_hgraph {
    int *dist; // has to given size of property type
    int *olddist; // has to given size of property type
    bool *uptd; // has to given size of property type
};

void alloc_extra_hgraph(HGraph &hgraph, int flag) {
    if(flag==0) hgraph.extra = (struct struct_hgraph *)malloc(sizeof(struct struct_hgraph));
    ((struct struct_hgraph *)hgraph.extra)->dist = (int *)malloc(sizeof(int) * hgraph.npoints);
}

void alloc_extra_graph(GGraph &graph, int flag) {
    struct struct_hgraph temp;
    if(flag==0) cudaMalloc((void **)&(graph.extra), sizeof(struct struct_hgraph));
    cudaMemcpy(&temp, ((struct struct_hgraph *)graph.extra), sizeof(struct struct_hgraph), cudaMemcpyDeviceToHost);
    cudaMalloc( (void **) &temp.olddist), sizeof(int *) * graph.npoints);
    cudaMalloc( (void **) &temp.uptd), sizeof(bool) * graph.npoints);
    if(cudaMemcpy(graph.extra, &temp, sizeof(struct struct_hgraph), cudaMemcpyHostToDevice) != cudaSuccess) printf("memcpyerror 0");
}

//GENERATED CUDA FOR SSSP EXAMPLE
#include "sssp.h"
cudaDeviceProp prop;
__device__ int changed = 0;

__global__ relaxgraph ( GGraph graph, int FALCX )
{
    int id= blockIdx.x * blockDim.x + threadIdx.x + FALCX;
    if(id < graph.npoints && ((struct struct_hgraph *)graph.extra)->uptd == true)
    {
        ((struct struct_hgraph *)graph.extra)->uptd[id] = false;
        int falcft0 = graph.index[id+1]-graph.index[id];
        int falcft1 = graph.index[id];
        int falcft2 = 0;falcft2<falcft0;falcft2++)(
            int ut0=2*(falcft1+falcft2);
            int ut1=graph.edges[ut0].ipe;
            int ut2=graph.edges[ut0+1].ipe;
            GMIN(&(((struct struct_hgraph *)graph.extra)->dist[ut1]),((struct struct_hgraph
            *)graph.extra)->dist[id]+ut2,**/changed);
        }//foreach
```c
__global__ reset ( GGraph graph, int FALCX )
{
    int id = blockIdx.x * blockDim.x + threadIdx.x + int FALCX;
    if (id < graph.npoints) {
        ((struct struct_hgraph *)((graph.extra))->dist[id] = 1234567890;
        ((struct struct_hgraph *)((graph.extra))->olddist[id] = 1234567890;
        ((struct struct_hgraph *)((graph.extra))->uptd[id] = false;
    }
}

__global__ reset1 ( GGraph graph, int FALCX )
{
    int id = blockIdx.x * blockDim.x + threadIdx.x + FALCX;
    if (id < graph.npoints) {
        if (((struct struct_hgraph *)((graph.extra))->dist[id] < ((struct struct_hgraph *)((graph.extra))->olddist[id]))
            ((struct struct_hgraph *)((graph.extra))->uptd[id] = true;
        ((struct struct_hgraph *)((graph.extra))->olddist[id] = ((struct struct_hgraph *)((graph.extra))->dist[id]);
    }
}

main ( int argc, char * argv[] )
{
    cudaGetDeviceProperties(&prop, 0);
    HGraph hgraph;
    GGraph graph;
    hgraph.read(argv[1]);
    int hosthgraph = 1;
    hgraph.extra = (struct struct_hgraph *)malloc(sizeof(struct struct_hgraph));
    alloc_extra_hgraph(hgraph, hosthgraph);
    hgraph.cloneGPU(graph, 0);
    int graphflag = 0;
    alloc_extra_graph(graph, graphflag);
    int TPB = findthreadsperblock(prop);
    int graphpointkernelblocks = findblocksize(graph, graph.npoints, TPB);
    for (int ii = 0; ii < graph.npoints; ii += graphpointkernelblocks * TPB) {
        reset<<<graphpointkernelblocks, TPB>>>(graph, ii);
    }
    cudaDeviceSynchronize();
    int falcvt1;
    falcvt1 = 0;
    struct struct_hgraph ftemp1;
    cudaMemcpy(&ftemp1, ((struct struct_hgraph *)((graph.extra))->dist), sizeof(struct struct_hgraph), cudaMemcpyDeviceToHost);
    if (cudaMemcpy(&((ftemp1.dist[0]), sizeof(int)), cudaMemcpyHostToDevice) != cudaSuccess)
        printf("memcpyerror 1");
    bool falcvt2;
    falcvt2 = true;
    if (cudaMemcpy(&((ftemp1.uptd[0]), sizeof(bool)), cudaMemcpyHostToDevice) != cudaSuccess)
        cudaMemcpyHostToDevice!
```
=cudaSuccess)printf("memcpyerror 1");
  while(1) {
    int falcvt3;
    falcvt3=0;
    if(cudaMemcpyToSymbol(changed,&(falcvt3),sizeof(int ),0,cudaMemcpyHostToDevice)!
             =cudaSuccess)printf("memcpyerror 2");
        for(int ii=0;ii<graph.npoints;ii+=graphpointkernelblocks*TPB){
          relaxgraph<<<graphpointkernelblocks,TPB>>>(graph,ii);
        }
    cudaDeviceSynchronize();
    int falcvt4;
    if(cudaMemcpyFromSymbol(&falcvt4,changed,sizeof(int ),0,cudaMemcpyDeviceToHost)!
             =cudaSuccess)printf("memcpyerror 3");
        if(falcvt4==0)break;
        for(int ii=0;ii<graph.npoints;ii+=graphpointkernelblocks*TPB){
          reset1<<<graphpointkernelblocks,TPB>>>(graph,ii);
        }
    cudaDeviceSynchronize();
  }
  struct struct_hgraph ftemp2;
  int *ftemp3 = (int *)malloc(sizeof(int )*graph.npoints);
  cudaMemcpy(&ftemp2,((struct struct_hgraph *)(graph.extra)),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
  if(cudaMemcpy((ftemp3),(ftemp2.dist),sizeof(int )*graph.npoints,cudaMemcpyDeviceToHost)!
               =cudaSuccess)printf("memcpyerror 4");
      for(int i=0;i<graph.npoints;i++)printf("%d \n",ftemp3[i]);
  }
int <GPU> changed=0;
void BFS(Point <GPU> p, Graph <GPU> graph) {
  foreach(t In p.outnbrs) {
    MIN(t.dist, p.dist + 1, graph.changed[0]);
  }
}

relaxgraph( Point <GPU> p, Graph <GPU> graph) {
  foreach(t In p.outnbrs) {
    MIN(t.dist, p.dist + graph.getWeight(p, t), graph.changed[0]);
  }
}

SSSPBFS(char *name) {
  Graph hgraph;
  hgraph.addPointProperty(dist, int);
  hgraph.addProperty(changed, int);
  hgraph.getType() <GPU> graph;
  hgraph.getType() <GPU> graph1;
  hgraph.addPointProperty(dist1, int);
  hgraph.read(name);
  graph = hgraph;
  graph1 = hgraph;
  foreach(t In graph.points)t.dist = 1234567890;
  foreach(t In graph1.points)t.dist = 1234567890;
  graph.points[0].dist = 0;
  graph1.points[0].dist = 0;
  parallel sections {
    section {
      while(1) {
        graph1.changed[0] = 0;
        foreach(t In graph1.points) BFS(t, graph1);
        if (graph1.changed[0] == 0) break;
      }
    }
    section {
      while(1) {
        graph.changed[0] = 0;
        foreach(t In graph.points) relaxgraph(t, graph);
        if (graph.changed[0] == 0) break;
      }
    }
  }
  hgraph.dist = graph.dist; // sssp dist
  hgraph.dist1 = graph1.dist; // bfs dist
  return;
}

int main(int argc, char *argv[]) {
  SSSPBFS(argv[1]);
}
#include "HGraph.h"
#include "HSet.h"
#include "thrust.cu"
#include<sys/time.h>
#include </usr/local/cuda/include/cuda.h>
#include </usr/local/cuda/include/cuda_runtime_api.h>
#include<unistd.h>
#include</usr/local/cuda-7.0/samples/0_Simple/simplePrintf/cuPrintf.cu>

struct struct_hgraph {
    int   *dist1 ;//has to given size of property type
    int nchanged;//user should enter one for reading function.
    int   *changed ;//has to given size of property type
    int   *dist ;//has to given size of property type
};

void read_hgraph_pptysize(HGraph &hgraph){
    printf("enter size of propery changed of hgraph ");
    //ENTER ONE(1) FOR THIS
    scanf("%d",&(((struct struct_hgraph  *)(hgraph.extra))->nchanged));
}

void alloc_extra_hgraph(HGraph &hgraph,int flag) {
    if(flag==0)hgraph.extra=(struct struct_hgraph  *)malloc(sizeof(struct struct_hgraph  ));
    (struct struct_hgraph  *)hgraph.extra)->dist1=(int  *)malloc(sizeof(int ) * hgraph.npoints) ;
    (struct struct_hgraph  *)hgraph.extra)->changed=(int  *)malloc(sizeof(int ) * ((struct
    struct_hgraph  *)(hgraph.extra))->nchanged) ;
    (struct struct_hgraph  *)hgraph.extra)->dist=(int  *)malloc(sizeof(int ) * hgraph.npoints) ;
}

void alloc_extra_graph(GGraph &graph,int flag) {
    cudaSetDevice(0);
    struct struct_hgraph  temp;
    if(flag==0)cudaMalloc((void **)&(graph.extra),sizeof(struct struct_hgraph  ));
    cudaMemcpy(&temp,((struct struct_hgraph  *)(graph.extra)),sizeof(struct struct_hgraph  ),cudaMemcpyDeviceToHost);
    cudaMalloc((void **)&( temp.changed),sizeof(int )* temp.nchanged);
    cudaMemcpy((void **),(temp.dist),sizeof(int )* graph.npoints);
    cudaMemcpy(graph.extra,&temp,sizeof(struct struct_hgraph  ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 0");
    cudaSetDevice(0);
}

void alloc_extra_graph1(GGraph &graph1,int flag) {
    cudaSetDevice(1);
    struct struct_hgraph  temp;
    if(flag==0)cudaMalloc((void **)((graph1.extra),sizeof(struct struct_hgraph  ));
    cudaMemcpy(&temp,((struct struct_hgraph  *)(graph1.extra)),sizeof(struct struct_hgraph  ),cudaMemcpyDeviceToHost);
    cudaMalloc((void **)((temp.changed),sizeof(int )* temp.nchanged);
    cudaMemcpy((void **)((temp.dist),sizeof(int )* graph1.npoints);
    cudaMemcpy(graph1.extra,&temp,sizeof(struct struct_hgraph  ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 1");
    cudaSetDevice(0);
}
CUDA FILE GENERATED
#include "multigpu.h"
cudaDeviceProp prop;
cudaDeviceProp prop1;
__device__ int  changed =0;
__global__  BFS ( GGraph  graph,int FALCX ) {
    int id= blockIdx.x * blockDim.x + threadIdx.x+FALCX;
    if(id < graph.npoints){
        int falcft0=graph.index[id+1]-graph.index[id];
        int falcft1=graph.index[id];
        for(int falcft2=0;falcft2<falcft0;falcft2++){
            int ut0=2*(falcft1+falcft2);
            int ut1=graph.edges[ut0].ipe;
            int ut2=graph.edges[ut0+1].ipe;
            GMIN(&(((struct struct_hgraph  *)(graph.extra))->dist[ut1]),((struct struct_hgraph  *)(graph.extra))->dist[id]+1,/**/((struct struct_hgraph  *)(graph.extra))->changed[0]);
        }//foreach
    }
}
__global__  relaxgraph ( GGraph  graph,int FALCX ) {
    int id= blockIdx.x * blockDim.x + threadIdx.x+FALCX;
    if(id < graph.npoints){
        int falcft3=graph.index[id+1]-graph.index[id];
        int falcft4=graph.index[id];
        for(int falcft5=0;falcft5<falcft3;falcft5++){
            int ut3=2*(falcft4+falcft5);
            int ut4=graph.edges[ut3].ipe;
            int ut5=graph.edges[ut3+1].ipe;
            GMIN(&(((struct struct_hgraph  *)(graph.extra))->dist[ut4]),((struct struct_hgraph  *)(graph.extra))->dist[id]+ut5,/**/((struct struct_hgraph  *)(graph.extra))->changed[0]);//rhs not null
        }//foreach
    }
}
__global__ void falctfun0(GGraph graph,int FALCX){
    int id= blockIdx.x * blockDim.x + threadIdx.x+FALCX;
    if (id < graph.npoints) {
        ((struct struct_hgraph  *)(graph.extra))->dist[id]=1234567890;
    }
}
__global__ void falctfun1(GGraph graph1,int FALCX){
    int id= blockIdx.x * blockDim.x + threadIdx.x+FALCX;
    if (id < graph1.npoints) {
        ((struct struct_hgraph  *)(graph1.extra))->dist[id]=1234567890;
    }
}
SSSPBFS ( char  *  name ) {
    HGraph  hgraph ;
GGraph graph;
GGraph graph1;
hgraph.read(name);
int hosthgraph=1;
hgraph.extra=(struct struct_hgraph *)malloc(sizeof(struct struct_hgraph ));
read_hgraph_pptysize(hgraph);//THIS FUNCTION READS SIZE OF PROPERTY

CHANGED AND ALLOCATE MEMORY FOR IT.user can enter 1.
alloc_extra_hgraph(hgraph,hosthgraph);
graph.cloneGPU(graph,0);
int TPB=findthreadsperblock(prop);
int graphpointkernelblocks=findblocksize(graph,graph.npoints,TPB);
int graphflag=0;
cudaSetDevice(0);
cudaMalloc((void **)(&graph.extra),sizeof(struct struct_hgraph ));
struct struct_hgraph ftemp1;
if(cudaMemcpy(&ftemp1,graph.extra,sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 2");
ftemp1.nchanged=(( struct struct_hgraph *)(hgraph.extra))->nchanged;//STORE SIZE OF CHANGED VARIABLE FOR GPUGRAPH graph
if(cudaMemcpy(graph.extra,&ftemp1,sizeof(struct struct_hgraph ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 3");
cudaSetDevice(0);
alloc_extra_graph(graph,graphflag);
int falcvt1;
graph.cloneGPU(graph1,1);
int TPB1=findthreadsperblock(prop1);
int graph1pointkernelblocks=findblocksize(graph1,graph1.npoints,TPB1);
int graph1flag=0;
cudaSetDevice(1);
cudaMalloc((void **)(&graph1.extra),sizeof(struct struct_hgraph ));
struct struct_hgraph ftemp4;
if(cudaMemcpy(&ftemp4,graph1.extra,sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 5");
temp4.nchanged=(( struct struct_hgraph *)(hgraph.extra))->nchanged;//COPY SIZE OF CHANGED VARIABLE FOR GPUGRAPH graph1
if(cudaMemcpy(graph1.extra,&ftemp4,sizeof(struct struct_hgraph ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 6");
cudaSetDevice(0);
for(int ii=0;ii<graph.npoints;ii+=graphpointkernelblocks*TPB){
    falcft0<<<graphpointkernelblocks,TPB>>>(graph,ii);
}
cudaDeviceSynchronize();cudaSetDevice(0);
cudaSetDevice(1);
for(int ii=0;ii<graph1.npoints;ii+=graph1pointkernelblocks*TPB1){
    falcft1<<<graph1pointkernelblocks,TPB1>>>(graph1,ii);
}
cudaDeviceSynchronize();cudaSetDevice(0);
int falcvt2;

falcvt2=0;
struct struct_hgraph ftemp5;
cudaMemcpy(&ftemp5,((struct struct_hgraph *)graph.extra),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
    if(cudaMemcpy(&(ftemp5.dist[0]),&(falcvt2),sizeof(int ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 7");
    int falcvt3;
    falcvt3=0;
    struct struct_hgraph ftemp6;
    cudaMemcpy(&ftemp6,((struct struct_hgraph *)graph1.extra),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
    if(cudaMemcpy(&(ftemp6.dist[0]),&(falcvt3),sizeof(int ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 8");

#pragma omp parallel sections
{
    #pragma omp sections
    {
        while(1) {
            int falcvt4;
            falcvt4=0;
            struct struct_hgraph ftemp7;
            cudaMemcpy(&ftemp7,((struct struct_hgraph *)graph1.extra),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
            if(cudaMemcpy(&(ftemp7.changed[0]),&(falcvt4),sizeof(int ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 9");
            cudaSetDevice(1);
            for(int ii=0;ii<graph1.npoints;ii+=graph1pointkernelblocks*TPB1){
                BFS<<<graph1pointkernelblocks,TPB1>>>(graph1,ii);
            }
            cudaMemcpy(&falcvt5,&(ftemp7->changed[0]),sizeof(int ),cudaMemcpyDeviceToHost);
            if( falcvt5==0 )break;
        }
    }
}

    #pragma omp sections
    {
        while(1) {
            int falcvt6;
            falcvt6=0;
            struct struct_hgraph ftemp8;
            cudaMemcpy(&ftemp8,((struct struct_hgraph *)graph.extra),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
            if(cudaMemcpy(&(ftemp8.changed[0]),&(falcvt6),sizeof(int ),cudaMemcpyHostToDevice)!=cudaSuccess)printf("memcpyerror 10");
            cudaSetDevice(0);
            for(int ii=0;ii<graph.npoints;ii+=graphpointkernelblocks*TPB){
                relaxgraph<<<graphpointkernelblocks,TPB>>>(graph,ii);
            }
            cudaMemcpy(&falcvt5,&(ftemp7->changed[0]),sizeof(int ),cudaMemcpyDeviceToHost);
            if(cudaMemcpy(&falcvt5,&(ftemp7->changed[0]),sizeof(int ),cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 11");
            cudaSetDevice(0);
        }
    }
}
int falvc7;
if(cudaMemcpy(&falcvt7,&(ftemp8.changed[0]),sizeof(int ),cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 13");
if( falcvt7==0 )break;
}
}
}
struct struct_hgraph ftemp10;
cudaMemcpy(&ftemp10,((struct struct_hgraph *)(graph.extra)),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
if(cudaMemcpy(((struct struct_hgraph *)(hgraph.extra))->dist),
(ftemp10.dist),sizeof(int)*hgraph.npoints,cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 14");
struct struct_hgraph ftemp11;
cudaMemcpy(&ftemp11,((struct struct_hgraph *)(graph1.extra)),sizeof(struct struct_hgraph ),cudaMemcpyDeviceToHost);
if(cudaMemcpy(((struct struct_hgraph *)(hgraph.extra))->dist1),
(ftemp11.dist),sizeof(int)*hgraph.npoints,cudaMemcpyDeviceToHost)!=cudaSuccess)printf("memcpyerror 15");
return ;
}
main ( int argc ,char * argv [ ] )
{
cudaGetDeviceProperties(&prop, 0);
cudaGetDeviceProperties(&prop1, 1);
SSSPBFS(argv[1]);
}
8.1. Extra content about Falcon DSL

```c
Linkset(int n1, int n2) {
    if (n1 > n2) parent[n2] = n1;
    else parent[n1] = n2;
}

FindSet(int n1) {
    if (parent[n1] == n1)
        parent[n1] = FindSet(parent[n1]);
    else return parent[n1];
}

Union(int n1, int n2) {
    t1 = FindSet(n1);
    t2 = FindSet(n2);
    LinkSet(t1, t2);
}
```

Fig. 24: Pseudo Code for Set

```c
struct collvec {
    int front, back, size, *elems;
};

createcoll(struct collvec coll, int n1) {
    coll.elems = allocate(n1);  // allocate
    coll.size = n1; coll.back = coll.front = 0;
}

getelem(struct collvec coll, int n1) {
    int t1 = coll.elems[coll.front];
    coll.front++;
    return t1;
}

pushelem(struct collvec coll, int n1) {
    if (coll.back == coll.size - 1) {
        coll.elems = realloc(coll.size * 2);
        coll.size *= 2;
    }
    coll.elems[coll.back] = n1;
    coll.back++;
    return;
}
```

Fig. 25: Pseudo Code for Collection based on dynamic arrays
Fig. 26: Usage of Single in DMR

```c
for(i=0;i<pred.size;i++)pred[i].owner=threadid;
global_barrier();
backoff=false;
for(i=0;i<pred.size;i++)
  if(pred.owner < threadid)backoff=true;break
if(pred.owner > threadid)pred.owner=threadid;
global_barrier();
for(i=0;i<pred.size;i++)
  if(pred.owner !=threadid)backoff=true;break
if(backoff==false){
  //statement to update cavity
} else { //abort
}
```

Fig. 27: Psuedo Code for Single on Collection in Fiugre 28

Fig. 28: Usage of Single in DMR

```c
for(i=0;i<pred.size;i++)pred[i].owner=threadid;
global_barrier();
backoff=false;
for(i=0;i<pred.size;i++)
  if(pred.owner < threadid)backoff=true;break
if(pred.owner > threadid)pred.owner=threadid;
global_barrier();
for(i=0;i<pred.size;i++)
  if(pred.owner !=threadid)backoff=true;break
if(backoff==false){
  //statement to update cavity
} else { //abort
}
```

Fig. 29: Psuedo Code for Single on Collection in Fiugre 28
Graph hgraph;  // Code pattern is similar to that of Galois Code. Refer to the same for any doubts/
struct node{
/*Full DSL code in Figure ??*/
Point(hgraph) n1;
int w;};
Collection pred[struct node];
pred.OrderByIntValue(w,10);
for_each(t In pred)relaxNode1(t,hgraph,pred);

Fig. 30: Falcon DSL code for Collection on Multicore-CPU

struct nodeIndexer:public std::unary_function<struct node,unsigned int>{
unsigned int operator()(const struct node &val)const{
unsigned int t=val.w/1024;
return t;};
struct Process0 {
template<typename Pusher0>
Process0(){};
void operator() (struct node &req,Pusher0 &pred){
relaxNode1(req,hgraph,pred);
}
};
using namespace Galois::WorkList;
typedef dChunkedFIFO<64> Chunk;
typedef OrderedByIntegerMetric<struct nodeIndexer,Chunk,10> OBIM;
Galois::InsertBag<struct node> pred;
Galois::for_each_local(pred,Process0(),Galois::wl<OBIM>());

Fig. 31: Code generated for Collection on Multicore-CPU
Graph hgraph;/*Code pattern is similar to that of Galois Code. Refer to the same for any doubts*/
struct workitem{
  Point (hgraph) src, (hgraph) dst;
  int weight, cur;};
Collection WL1[struct workitem],WL2[struct workitem],MST[struct workitem];
Collection *current[struct workitem],*next[struct workitem];
Set hset[Point(hgraph)];
findLightest(struct workitem req,Graph hgraph,Collection pred [struct workitem],Collection next[struct workitem]){
  Point (hgraph) src=req.src;
  int cur=req.cur;
  foreach (t ( + cur) In src.outnbrs){
    int weight=hgraph.getWeight(src,t);
    if(hset.find(src)!=hset.find(t)){
      struct workitem tt;
      tt.src=src;tt.dst=t;tt.weight=weight;tt.cur=cur;
      next.add(tt);
      while(weight < (old=hgraph.points[rep].minedge))MIN(rep.minedge,weight,ch); }
    ++cur;
  }
}
void fun1( ){
collection temp[struct workitem];
// now fill WL1(current) with all points
while(1){
  foreach(t In *current)findLightest(t,hgraph,*current,*next);
  temp=current;current=next;next=temp;//swap current and next worklist
  foreach(t In *current) //union added elements in next,uses Collection variable MST to store edges in MST.
    UnionComponents(t,hgraph,*current,*next);
  if(!(*current).empty())break;//reached fixpoint.mst formed
}
}

Fig. 32: foreach with advance expression used in Falcon MST CPU code
The code snippet provided is a implementation of the PageRank algorithm on a GPU. It uses a graph manipulation language called Falcon. Here is a breakdown of the code:

```c
float <GPU>diff = 0.0; //atomicAdd on CUDA can be done by compile time flag -DFLOAT now
pagerank(Point<GPU> t, Graph<GPU> graph, int max, float e, float d) {
    float diff_prv = 0.0;
    float val = 0.0;
    float S1 = 0.0;
    foreach(p In t.innbrs) {
        S1 = S1 + (p.pagerank/p.outDegree());
    }
    val = ((1-d)/graph.npoints) + (d*S1);
    diff_prv = abs(val-t.pagerank);
    t.pageranknext = val;
    ADD(diff, diff_prv); //atomicAdd an on float, enabled by nvcc compiler(cuda 6.5) flag -DFLOAT
}
main(int argc, char *argv[]) {
    int max;
    float e, d;
    Graph hgraph;
    hgraph.addProperty(pagerank, float);
    hgraph.addProperty(pageranknext, float);
    hgraph.getType() <GPU> graph;
    hgraph.read(argv[1]);
    graph = hgraph;
    printf("Enter values max, e and d");
    scanf("%d%f", &max, &e, &d);
    foreach(t In graph.points) t.pagerank = 1/graph.npoints;
    int itr = 0;
    while(itr < max) {
        diff = 0;
        ++itr;
        foreach(t In graph.points) pagerank(t, graph, max, e, d);
        if(diff < e) break;
        foreach(t In graph.points) t.pagerank = t.pageranknext;
    }
}
```

The code starts by initializing variables for the PageRank algorithm. It then defines the `pagerank` function which takes a point, a graph, a maximum number of iterations, and parameters `e` and `d`. The `main` function reads input parameters and initializes a graph. It then iterates through the graph points, updating their PageRank values according to the PageRank formula. The `printf` and `scanf` functions are used to read input values from the user.

---

**Fig. 33:** Page ranking algorithm in GPU

---

int <GPU> changed=0,hchanged,hlevel,<GPU> level;
void reset (Point <GPU> t, Graph <GPU> graph){
t.dist=1234567890;
t.pred=-1;
t.sigma=t.delta=t.bc=0.0;;
}
void BFS(Point <GPU> p,Graph <GPU> graph){//This is not optimized BFS, Optimized atomic free BFS mentioned later in document
int ch;
foreach(t In p.OutNbrs){
MIN(t.dist,p.dist+1,changed);
if(t.dist==p.dist+1)t.pred=p;
}
void sigma1(Point <GPU> p,Graph <GPU> graph, int lev){
foreach(t In p.OutNbrs) if(t.dist==0) t.sigma=1;
}
void sigma(Point <GPU> p,Graph <GPU> graph, int lev){
foreach(t In p.OutNbrs)if(t.dist==lev+1) ADD(t.sigma,p.sigma); /*atomicAdd on float, enabled by compiletime flag -DFLOAT*/
}
void delta1(Point <GPU> p,Graph <GPU> graph, int lev){
foreach(t In p.OutNbrs)if(t.dist==level) t.delta=0;
}
void delta(Point <GPU> p,Graph <GPU> graph, int lev){
float temp=0; foreach(t In p.OutNbrs)if(t.dist==lev+1)temp=temp+p.sigma/t.sigma*(1+ t.delta);
p.delta=temp+p.delta
}
void BC(Point <GPU> p,Graph <GPU> graph){
p.bc=p.bc+p.delta;
}
}

Fig. 34: Betweeness Centrality DSL code part1
```c
int <GPU> changed; int main(int argc, char *name[]){
    Graph hgraph;
    hgraph.addPointProperty(dist,int);//All the properties of Point can be put in a structure
    hgraph.addPointProperty(pred,int);
    hgraph.addPointProperty(sigma,float);
    hgraph.addPointProperty(delta,float);
    hgraph.addPointProperty(bc,float);
    hgraph.getType() <GPU> graph;
    hgraph.read(name[1]);
    graph=hgraph;
    foreach(t In graph.points)t.bc=0.0;
    for(int i=0;i<graph.npoints;i++)//assumes do BC for all, can be reduced to few iterations
        foreach(t In graph.points)reset(t,graph);//reset properties
        graph.points[i].dist=0;
        graph.points[i].pred=i;hlevel=0;
        while(1){
            changed=0;
            foreach(t In graph.points)BFS(t,graph);//compute new BFS distance
            if(changed==0)break;
            else hlevel++;
        }
    int temp=1;
    foreach(t In graph.points)sigma1(t,graph,temp);
    while(temp < hlevel){
        foreach(t In graph.points)sigma(t,graph,temp);
        temp++;
    }
    temp=hlevel;
    foreach(t In graph.points)delta1(t,graph,temp);
    temp--;
    while(temp >0){
        foreach(t In graph.points)delta(t,graph,temp);
        temp--;
    }
    foreach(t In graph.points)BC(t,graph);
}
```

Fig. 35: Betweeness Centrality DSL code part2
int <GPU> changed=0;<GPU>coll2size;

BFS(Point <GPU> p,Graph <GPU> graph,Collection <GPU> *curr[Point(graph)],Collection<GPU> *next[Point(graph)]) {
    foreach(t In p.outnbrs){
        MIN(t.dist,p.dist+1,changed);
        if(t.dist==p.dist+1){
            next.add(t);
            ADD(coll2size,1);
        }
    }
}

main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.read(name[1]);
    hgraph.getType() <GPU> graph;
    Collection(coll1[Point(graph)],coll2[Point(graph)],*ptr1[Point(graph)],*ptr2[Point(graph)],*ptr3[Point(graph)]);
    graph=hgraph;
    foreach(t In graph.points)t.dist=1234567890;
    graph.points[0].dist=0;
    ptr1=&coll1;
    ptr2=&coll2;
    count=0;  coll1.add(graph.points[0]);
    int size=1;
    while(1){
        changed=0;
        ptr1->size=size;
        foreach(t In *ptr1)BFS(t,graph.ptr1,ptr2);
        ptr3=ptr1; ptr1=ptr2; ptr2=ptr3; /*swap worklists*/
        ptr1->size=coll2size;
        ptr2->size=hgraph.npoints;
        coll2size=0;
        if(changed==0)break;
    }
    hgraph.dist=graph.dist;
    for(int i=0;i<hgraph.npoints;i++)
        printf("%d\n",hgraph.points[i].dist);
    return;
}

Fig. 36: BFS code using Collection for GPU in Falcon.
void drefine(struct node t, Graph graph, Collection current[struct node], Collection collnext[struct node])
2  struct cent cent_new;
3  Collection pred[struct node (graph)], frontier[struct node (graph)], post[struct node (graph) ], conntria1[struct node (graph)], conntria2[struct node (graph)];
4  Collection conn1[Point (graph)], conn2[Point (graph)], conn3[Point (graph)], conn4[Point (graph)];
5  struct node (graph) ceelem = t;
6  if (t.isdel) && (t.isbad){
7    int cebot = t.obtuse, itr = 0;
8    while (cebot < 3 && ++itr < 50 && ceelem < graph.ntriangle) {
9      ceelem = getOpposite(ceelem, graph, cebot);
10     if (ceelem < graph.ntriangle) cebot = ceelem.obtuse;
11   } if (ceelem >= graph.ntriangle || ceelem.isdel{
12     ceelem = t; cebot = ceelem.obtuse;
13   } cent_new = getCenter(ceelem, graph);
14  P1, (graph)P0;
15  P1.x = cent_new.centerx; P1.y = cent_new.centery;
16  pred.add(ceelem), frontier.add(ceelem);
17  while (frontier.getSize() > 0) {
18    struct node (graph) curr;
19    frontier.del(curr);
20    if (curr >= graph.ntriangle) break;
21    int dims;
23    for (int i = 0; i < dims; i++) {
24      struct node (graph) next;
25      next = curr.neighbours[i];
26      if (next >= graph.ntriangle) break;
27      if (next >= graph.ntriangle) break; // modi
28      if (next.isdel) continue;
29      int nextdims;
31      if (!next! = ceelem && dims == 2 && nextdims == 2 && inCircumCircle(P1, next, graph)) {
32        if (nextdims == 2 && dims == 2) {
33          for (int i = 0; i < pred.size; i++) collnext.add(pred.del());
34          if (next.size == 0) pred.size = 0; conn1.size = 0; conn2.size = 0; conntria1.size = 0; conntria2.size = 0; ceelem = next;
35          cebot = ceelem.obtuse; itr = 0;
36          while (cebot < 3 && ++itr < 50 && ceelem < graph.ntriangle) {
37            ceelem = getOpposite(ceelem, graph, cebot);
38            if (ceelem < graph.ntriangle) {
39              cebot = ceelem.obtuse;
40            } else {
41              cent_new = getCenter(ceelem, graph);
42              P1.x = cent_new.centerx; P1.y = cent_new.centery;
43              pred.add(ceelem); frontier.add(ceelem);
44            } } else {
45              for (jj = 0; jj < pred.getSize(); jj++) if (pred[jj] == next) break;
46              if (jj == pred.getSize()) {
47                pred.add(next);
48                frontier.add(next);
49              } } }
50  }

Fig. 37: DMR Mesh refinement CPU code Part1
```c
else {
    int pi1=curr.neighedgestart[i];
    if(pi1>=3)continue;
    Point (graph)p1=curr.nodes[pi1],(graph)p2=curr.nodes[(pi1+1)]
    unsigned jj;
    for(jj=0;jj<conn1.getSize();jj+=1)if(conn1[jj]==p1 & & conn2[jj]==p2)break;
    if(jj==conn1.getSize()){
        conn1.add(p1);conn2.add(p2);conntria1.add(curr);conntria2.add(next);
    }
}
//end if line 6
single(pred){
    if( !( t.isdel) && (t.isbad)){
        Point (graph) p0;
        int dims;
        p0=graph.addPoint(cent_new.centerx,cent_new.centery);
        if(dims==2){
            struct node (graph)tr1,(graph)tr2;
            tr1=addTriangle1(graph,p0,inv_point);
            tr2=addTriangle1(graph,p0,inv_point);
        }
        for (unsigned ii = 0; ii < conn1.getSize(); ii +=1) { Point (graph) p1=conn1[ii], p2=conn2[ii];
            struct node (graph) connsrc=conntria1[ii], (graph) conndst=conntria2[ii], (graph)
            newtri=addTriangle(graph,p0,p1,p2);
            int jj1; for (jj1 = 0; jj1 < pred.getSize(); jj1+=1) { if (pred[jj1] == conndst) break;
                jj1++;
            } struct node (graph) newconn;
            if(jj1==pred.getSize())newconn=conndst;
            else newconn=connsrc;
            int imnbr=0;
            newtri.neighbours[0]=newconn;newtri.neighedgestart[0]=1;
            imnbr++;
            for (unsigned jj = 0; jj < post.getSize(); ++jj) {
                int commonedgestart= adjacent(post[jj], newtri,graph);
                if (commonedgestart < 3 && imnbr<3){
                    newtri.neighbours[imnbr]=post[jj];newtri.neighedgestart[imnbr]=commonedgestart;
                    imnbr++;
                }
            }
        }
    }
}
/*end if line 61*/ else { collnext.add(t); }
}
```

Fig. 38: Mesh Refinement CPU Code Part2
struct node{
  struct_rec node neighbours[3];  //recursive structure
  Point neighedgestart[3],nodes[3];
  int isbad, isdel, obtuse, owner, dims, index;
}; //fields of property triangle
/*Two worklists(WL1,WL2) to store bad triangles, three worklists pointers to swap between WL1,WL2
in each iteration*/
Collection WL1[struct node],WL2[struct node], *current[struct node],*collnext[struct node],*colltemp[struct
node];
int main(int argc,char *argv[]){
  int ntria;
  Graph hgraph;
  hgraph.addProperty(triangle ,struct node);  //attach property triangle to hgraph
  hgraph.readNodes(argv[1],2);  //reads Mesh Points.class function
  readTriangle(hgraph,argv[1]);  //reads Mesh Triangles.User written function
  foreach(t In hgraph.triangle)dinit(t,hgraph);  //This finds initial bad triangles and adds it to WL1
  foreach(t In hgraph.triangle)findneighbours(t,hgraph);  //Find neighbours for each triangle
  /*In each iterations one work list has bad triangles,which will be processed,new bad triangles will put in
other worklist. Then swap the worklist using worklist pointers current,collnext and colltemp.*/
  current=&WL1;collnext=&WL2;
  do{
    foreach(t In *current)drefine(t,hgraph,*current,*collnext);  //process bad triangles, find new bad
    triangles
    colltemp=current;current=collnext;collnext=current; //swap worklists
    if(*current).empty()break;
  }while(1);
  foreach(t In hgraph.triangle)verify(t,hgraph);
}

Fig. 39: Mesh Refinement CPU main function
8.2. Unoptimized SSSP in Falcon

```c
int <GPU> changed=0;
relaxgraph( Point <GPU> p, Graph <GPU> graph) {
    foreach( t In p.outnbrs )
        MIN(t.dist,p.dist+graph.getWeight(p,t),changed);
}
SSSP(char *name) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.getType() <GPU> graph;
    hgraph.read(name);
    graph=hgraph;
    foreach( t In graph.points ) t.dist=1234567890;
    graph.points[0].dist=0;
    while( ) {
        changed=0;
        foreach( t In graph.points ) relaxgraph(t,graph);
        if(changed==0) break;
    }
    hgraph.dist=graph.dist;
    for(int i=0;i<hgraph.npoints;i++)
        printf("%d\n",hgraph.points[i].dist);
    return;
}
main(int argc, char *argv[]) {
    SSSP(argc[1]);
}
```

Fig. 40: SSSP code for GPU in Falcon

Figure 40 shows the SSSP example in Falcon for the GPU. This code is not optimized when compared to the code given in next section. Here we add only one property dist to each point in the CPU Graph object hgraph and the GPU Graph object graph also gets the same property as it is declared with the compile time function getType(), which inherits the dynamic properties of the hgraph to the graph. Then we make distance of GPU Graph objects source node zero. Then we keep on relaxing the nodes using the relaxgraph() function. Before calling the relaxgraph() function the GPU variable changed is made zero and its value is checked after exit from relaxgraph() function. If it is zero, indicating the fix point, we exit from the while loop and then we copy distance value from GPU to CPU.

One problem with this algorithm is that when we call the relaxgraph() function first time, all the points will do the atomic operation MIN, but value of dist will change only for neighbours of source node(in first invocation). So this code do MIN for all nodes(in all iterations), but there will be change in value of dist for only few nodes in each invocation. So this code will produce correct output, but running time can reduced with some optimizations.

Figure 41 shows the SSSP example in Falcon for the CPU. This code is similar to the code in Figure 40. Here we add only one property dist to each point in the CPU Graph object hgraph. There is no need of a GPU Graph object, as all computation is on the CPU. Then we make distance of CPU Graph objects source node zero. Then we keep on relaxing the nodes using the relaxgraph() function. Before calling the relaxgraph() function the CPU variable changed is made...
int changed=0;

relaxgraph(Point p, Graph graph) {
    foreach (t In p.outnbrs) {
        MIN(t.dist, p.dist + graph.getWeight(p,t), changed);
    }
}

SSSP(char *name) {
    Graph hgraph;
    hgraph.addPointProperty(dist, int);
    hgraph.read(name);
    foreach (t In hgraph.points) t.dist = 1234567890;
    hgraph.points[0].dist = 0;
    while (1) {
        changed = 0;
        foreach (t In hgraph.points) relaxgraph(t, hgraph);
        if (changed == 0) break;
        for (int i = 0; i < hgraph.npoints; i++)
            printf("%d
", hgraph.points[i].dist);
    return;
}

main(int argc, char *argv[]) {
    SSSP(argv[1]);
}

Fig. 41: SSSP code for CPU in Falcon

zero and its value is checked after exit from relaxgraph() function. If it is zero, indicating the fix point, we exit from the while loop.

This code also do atomic MIN operation for all nodes (in all iterations), but there will be change in value of dist for only few nodes in each invocation. So this code will produce correct output, but running time can reduced with some optimizations. when compared to code given in next section.
int <GPU> changed = 0;

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

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

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

main(int argc, char *argv[]) {
    Graph hgraph; // graph on CPU
    hgraph.addPointProperty(dist, int);
    hgraph.addPointProperty(upt, bool);
    hgraph.addPointProperty(oldDist, int);
    hgraph.getType() <GPU> graph; // graph on GPU
    hgraph.read(argv[1]); // read graph on CPU
    graph = hgraph; // copy graph to GPU
    // initialize GPU graph object properties
    foreach (t In graph.points) reset(t, graph);
    graph.points[0].dist = 0; // source has dist 0
    graph.points[0].uptd = true;
    while (1) {
        changed = 0; // keep relaxing on GPU
        foreach(t In graph.points) (t.uptd) relaxGraph(t, graph);
        if(changed == 0) break; // reached fix point
        foreach(t In graph.points) reset1(t, graph);
    }
    hgraph.dist = graph.dist; // copy all points dist to CPU
    for(int i = 0; i < hgraph.npoints; ++i)
        printf("i=%d dist=%d\n", i, hgraph.points[i].dist);
}

Fig. 42: Optimized SSSP code for GPU in Falcon

8.3. Optimized SSSP in Falcon

Here the code is optimized to make sure that only the nodes on which there is a need to take atomic MIN operation will do the same. To be more precise, if inside a GPU thread warp group all threads have the uptd false, then none of them will execute it, otherwise all will execute, still running time reduces. So this code will produce correct output, but takes less time than that of algorithm given in 8.2.

The Figure 43 is the CPU code for optimized variant shown in the Figure 42. Here again there is no need to create two graph object as the entire processing is done on the CPU. This code also makes sure that atomic MIN operation is performed only by those nodes which are actually supposed to do it. So it produces correct output, in short time compared to the code given in the Section 8.2.
int changed = 0;

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

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

reset1(Point t, Graph graph) {
    if(t.uptd==true && t.dist==t.olddist)t.uptd=false;
    t.olddist=t.dist;
}

main(int argc, char *argv[]) {
    Graph hgraph; // graph on CPU
    hgraph.addPointProperty(dist, int);
    hgraph.addPointProperty(upt, bool);
    hgraph.addPointProperty(olddist, int);
    hgraph.read(argv[1]); // read graph on CPU
    // initialize CPU graph object properties
    foreach (t In hgraph.points) reset(t, hgraph);
    hgraph.points[0].dist = 0; // source has dist 0
    hgraph.points[0].uptd=true;
    while ( ) {
        changed = 0; //keep relaxing on CPU
        foreach(t In hgraph.points)
            (t.uptd) relaxgraph(t, hgraph);
        if(changed == 0)break;//reached fix point
        foreach(t In hgraph.points) reset1(t, graph)
    }
    for(int i = 0; i < hgraph.npoints; ++i)
        printf("i=%d dist=%d\n", i, hgraph.points[i].dist);
}

Fig. 43: Optimized SSSP code for CPU in Falcon
8.4. Unoptimized BFS in Falcon

```c
int <GPU> changed=0;

BFS(Point <GPU> p, Graph <GPU> graph) {
    foreach (t In p.outnbrs) MIN(t.dist, p.dist+1, changed);
}

main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addProperty(dist, int);
    hgraph.read(name[1]);
    hgraph.getType() <GPU> graph;
    foreach (t In graph.points) t.dist=1234567890;
    graph.points[0].dist=0;
    while ( ){
        changed=0;
        foreach (t In graph.points) BFS(t, graph);
        if(changed==0)break;
    }
    hgraph.dist=graph.dist;
    for(int i=0;i<hgraph.npoints;i++)
        printf("%d\n", hgraph.points[i].dist);
    return;
}
```

Fig. 44: Unoptimized BFS code for GPU in Falcon

---

```c
int changed=0;

BFS(Point p, Graph graph) {
    foreach (t In p.outnbrs) MIN(t.dist, p.dist+1, changed);
}

main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addProperty(dist, int);
    hgraph.read(name[1]);
    foreach (t In hgraph.points) t.dist=1234567890;
    hgraph.points[0].dist=0;
    while ( ){
        changed=0;
        foreach (t In hgraph.points) BFS(t, hgraph);
        if(changed==0)break;
    }
    for(int i=0;i<hgraph.npoints;i++)
        printf("%d\n", hgraph.points[i].dist);
    return;
}
```

Fig. 45: Unoptimized BFS code for CPU in Falcon

Figure 44 shows the BFS code for GPU in Falcon. Here we create two graph objects one each for the GPU (graph) and the CPU (hgraph). We read the Graph using CPU Graph object and
copy it to GPU Graph object. We initialize dist property of all GPU Graph object points to a very large value. Then we make GPU Graph objects source node dist value zero. The we keep on calling the relaxgraph() function and reduce the dist value of each node using atomic MIN function. We initialize changed variable to zero, before calling the relaxgraph() and check its value after returing from relaxgraph(). If value of changed variable is zero after exit from the relaxgraph() function, we exit the while loop as this indicate a fixed-point.

Figure 45 shows the BFS code for CPU in Falcon. Here all computations take place in GPU so only one Graph object on CPU is created. The CPU objects Point property dist for all points is initialized to a large value by reset() function. Then we make dist value of source node zero. Then we keep on calling relaxgraph() function until there is no change to the dist value of any node. This is done by checking the value of the changed variable which is made zero just before calling relaxgraph() function. If the value of the changed variable is zero after returing from relaxgraph(), indicating a fixed point we exit from while loop.
8.5. Optimized BFS in Falcon

```c
int changed=0,lev=0;
BFS(Point p,Graph graph) {
    foreach (t In p.outnbrs) MIN(t.dist,p.dist+1,changed);
}
main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.read(name[1]);
    foreach (t In hgraph.points) t.dist=1234567890;
    hgraph.points[0].dist=0;
    while ( ) {
        changed=0;
        foreach (t In hgraph.points) (t.dist==lev) BFS(t,hgraph);
        if(changed==0) break;
        lev++;
    }
    for(int i=0; i<hgraph.npoints; i++) printf("%d\n",hgraph.points[i].dist);
    return;
}
```

Fig. 46: Optimized BFS code for CPU in Falcon

```c
int <GPU> changed=0,lev=0;
BFS(Point <GPU> p,Graph <GPU> graph) {
    foreach (t In p.outnbrs) MIN(t.dist,p.dist+1,changed);
}
main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.read(name[1]);
    hgraph.getType() <GPU> graph;
    graph=hgraph;
    foreach (t In graph.points) t.dist=1234567890;
    graph.points[0].dist=0;
    while ( ) {
        changed=0;
        foreach (t In graph.points) (t.dist==lev) BFS(t,graph);
        if(changed==0) break;
        lev++;
    }
    hgraph.dist=graph.dist;
    for(int i=0; i<hgraph.npoints; i++) printf("%d\n",hgraph.points[i].dist);
    return;
}
```

Fig. 47: Optimized BFS code for GPU in Falcon
Figure 47 shows the optimized BFS code for GPU in Falcon. There is not much difference from code for BFS given in Figure 44. Here code is optimized for running time. In first invocation of \textit{relaxgraph()} only source node will reduce \textit{dist} value of its neighbours. If we say in other words the \textit{i}th invocation will reduce distance of neighbours of nodes whose distance value is \((i-1)\). So we make foreach call to have a conditional to make sure that no unwanted computation take place. This will produce correct output and will run faster. Same optimization is done in Figure 46 if you compare with the code in Figure 45. In both the codes shown in this section, to perform this optimization we used a variable \textit{lev}(added in global declaration of Line 1), which stores current level of BFS traversal.
8.6. Atomic free BFS in Falcon

Figures 49 and 48 atomic free variant of BFS in Falcon. As both the algorithms do level by level traversal, in each iteration i (0 to N-1) the nodes which has a bfs distance i, will write to its neighbours value i+1. So in each iteration all threads write the same value, so atomic operation is not needed. This is the best possible code in Falcon and auto generated code outperforms codes in LonestarGPU, Totem, Galois and GreenMarl.

```c
int <GPU> changed=0, lev=0;

BFS(Point <GPU> p, Graph <GPU> graph) {
    foreach (t In p.outnbrs) {
        if (t.dist > p.dist + 1) {
            t.dist = p.dist + 1; changed = 1;
        }
    }
}

main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addPointProperty(dist, int);
    hgraph.read(name[1]);
    hgraph.setType() <GPU> graph;
    graph = hgraph;
    foreach(t In graph.points) t.dist = 1234567890;
    graph.points[0].dist = 0;
    while (1) {
        changed = 0;
        foreach (t In graph.points) (t.dist == lev) BFS(t, graph);
        if (changed == 0) break;
        lev++;
    }
    hgraph.dist = graph.dist;
    for(int i=0;i<hgraph.npoints;i++) printf("%d\n", hgraph.points[i].dist);
    return;
}
```

Fig. 48: Atomic free BFS code for GPU in Falcon
int changed=0,lev=0;

BFS(Point p,Graph graph) {
    foreach (t In p.outnbrs) {
        if (t.dist＞p.dist+1) {
            t.dist=p.dist+1; changed=1;
        }
    }
}

main(int argc, char *name[]) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.read(name[1]);
    foreach (t In hgraph.points)t.dist=1234567890;
    hgraph.points[0].dist=0;
    while (1) {
        changed=0;
        foreach (t In hgraph.points)(t.dist==lev)BFS(t,hgraph);
        if(changed==0)break;
        lev++;
    }
    for(int i=0;i<hgraph.npoints;i++)printf("%d\n",hgraph.points[i].dist);
    return;
}

Fig. 49: Atomic free BFS code for CPU in Falcon
8.6.1. Multi GPU code in Falcon. The code in Figure 50 shows how SSSP and BFS is computed at the same time using parallel sections statement of Falcon. One Important point to be noted here is how changed variable is used in the code. If we declare changed as shown in Line 1 of Figure 50, it will be allocated in GPU device 0. So to have changed to appear in each device it is added as a Graph Property in Line 15. For each property added using addProperty(), Falcon generates a function which read Property size, and also call to that function is inserted before allocating the property. So for GPU graph objects graph1 and graph changed will be allocated using cudaMalloc on appropriate device, as each memory operation and kernel call will be preceded by cudaSetDevice(), with GPU number of the graph object as argument. Now CPU graph object is added with another Point property dist1, so that it can store both BFS and SSSP distance using dist1 and dist. Then BFS and SSSP are computed using parallel sections statement of Falcon. The parallel sections statement will be converted to OpenMP parallel sections statement. This code can be extended by adding another GPU graph object, which can be used to compute MST on 3rd GPU in parallel.
//int <GPU> changed=0,hchanged
void BFS(Point <GPU> p,Graph <GPU> graph) {
    foreach (t In p.outnbrs) {
        MIN(t.dist,p.dist+1,graph.changed[0]);
    }
}

relaxgraph(Point <GPU> p,Graph <GPU> graph) {
    foreach (t In p.outnbrs) {
        MIN(t.dist,p.dist+graph.getWeight(p,t),graph.changed[0]);
    }
}

SSSPBFS(char *name) {
    Graph hgraph;
    hgraph.addPointProperty(dist,int);
    hgraph.addProperty(changed,int);
    hgraph.getType() <GPU> graph;
    hgraph.getType() <GPU> graph1;
    hgraph.addPointProperty(dist1,int);
    hgraph.read(name);
    graph=hgraph;
    graph1=hgraph;
    foreach (t In graph.points)t.dist=1234567890;
    foreach (t In graph1.points)t.dist=1234567890;
    graph.points[0].dist=0;
    graph1.points[0].dist=0;
    parallel sections {
        section {
            while(1){
                graph1.changed[0]=0;
                foreach (t In graph1.points)BFS(t,graph1);
                if(graph1.changed[0]==0) break;
            }
        }
        section {
            while(1){
                graph.changed[0]=0;
                foreach (t In graph.points)relaxgraph(t,graph);
                if(graph.changed[0]==0) break;
            }
        }
    }
    hgraph.dist=graph.dist;//sssp dist
    hgraph.dist1=graph1.dist;//bfs dist
    return;
}

int main(int argc,char *argv[]) {
    SSSPBFS(argv[1]);
}

Fig. 50: Multi-GPU BFS and SSSP in Falcon.
8.7. MST code for GPU in Falcon

The code in this section is shown for GPU (Figures 51 and 52). Here code assumes that graph is in symmetric format. Falcon has a method available in Graph class to convert a asymmetric graph to symmetric and vice versa. If it was not symmetric some changes are needed in algorithm such as removing code from Lines 17 to 23 of Figure 52.

```c
struct node{
    int lock, weight;
    Point set, src, dst;
};

int hchanged, changed;

void reset(Point <GPU> p, Graph<GPU> graph, Set set[Point(graph)]) {
    p.minppty.set.reset();//reset sets value to with MAX_INT
    p.minppty.src.reset();//replaced with reset()
    p.minppty.dst.reset();//replaced with reset()
    p.minppty.weight=99999999;
    p.minedge=99999999;
    p.minppty.lock=0;
}

void minset(Point <GPU> p, Graph<GPU> graph, Set set[Point(graph)]) {
    int ch;
    Point (graph) t1, t2;
    foreach (t in p.outnbrs) {
        t1=set.find(p);
        p.minedge=99999999;
        t2=set.find(t);
        if (t1!=t2) {
            MIN(t1.minppty.weight, graph.getWeight(p,t), ch);//find minimum edge out going from t1
            MIN(t2.minppty.weight, graph.getWeight(p,t), ch);//find minimum edge out going from t2.
        }
    }
}

void mstunion(Point <GPU> p, Graph<GPU> graph, Set set[Point(graph)]) {
    Point (graph)t1, (graph)t2;
    int t3, t4;
    t1=set.find(p);
    t2=t1.minppty.set; t3=t1.minppty.lock; t4=t2.minppty.lock;
    if (t1!=t2 && t3==t4) {
        set.Union(t1,t2);
        changed=1;
    }
}

initmark(Edge <GPU> e, Graph<GPU> graph {
    e.mark=99999999;
})
```

Fig. 51: Boruvka MST GPU part1
void Minedge(Point <GPU> p, Graph<GPU> graph, Set<Point(graph)> set) {
    Point(graph) t1, (graph)t2;
    int t3;
    Edge (graph) e;
    foreach (t In p.outnbrs) {
        t1=set.find(p);
        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; // add edge to mst
                t1.minppty.src=p; t1.minppty.dst=t; t1.minppty.weight=t3; t1.minppty.set=t2;
            }
            if (t3==t2.minppty.weight) {
                single (t2.minppty.lock)
                e=graph.getedge(p, t);
                e.mark=true; // add edge to mst
                t2.minppty.src=p; t2.minppty.dst=t; t2.minppty.weight=t3; t2.minppty.set=t2;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    Graph hgraph;
    hgraph.addPointProperty(minppty, struct node);
    hgraph.addEdgeProperty(mark, bool);
    hgraph.addNodeProperty(minedge, int);
    hgraph.getType() <GPU> graph;
    hgraph.read(argv[1]);
    Set hset[Point(hgraph)], set[Point(graph)];
    graph=hgraph;
    set=hset;
    foreach(t In graph.edges)initmark(t, graph);
    while (1) {
        changed=0;
        foreach(t In graph.points)reset(t, graph, set);
        foreach(t In graph.points)minset(t, graph, set);
        foreach(t In graph.points)Minedge(t, graph, set);
        foreach(t In graph.points)mstunion(t, graph, set);
        if (changed==0) break;
    }
    unsigned int mstsum=0;
    hgraph.mark=graph.mark;
    unsigned long int mst=0;
    foreach(t In hgraph.edges) {
        if (t.mark==1) mst=mst+t.weight;
    }
}

Fig. 52: Boruvka MST on GPU code part2
8.8. MST code for CPU in Falcon

This algorithm is a Worklist based algorithm which uses \texttt{FalconCollection} data type. The entire code is put in two Figures (Figure 53 and Figure 54). The \texttt{Collection} gets converted to \texttt{Galois::InsertBag} data structure, which is a worklist. This code will be converted to a code similar to the one which can be found in Galois-2.2 Boruvka MST code.
Graph hgraph;
Set hset[Point(hgraph)];
struct node{ Point (hgraph) src,Point (hgraph) dst;
  int weight;
};
int glimit,int bcnt;
struct workitem{
  Point (hgraph) src,Point (hgraph) dst;
  int weight,int cur;
};
Collection WL1[struct workitem],WL2[struct workitem],WL3[struct workitem],WL4[struct workitem];
Collection mst[struct node],*current[struct workitem],*next[struct workitem],*pending[struct workitem];
Collection *temp[struct workitem];

findLightest(struct workitem req,Graph hgraph,Collection pred [struct workitem],Collection next[struct workitem],Collection pending[struct workitem],int useLimit,Set hset[Point(hgraph)]) {
  struct workitem req1=req;
  Point (hgraph) src=req1.src;
  int cur=req1.cur;
  foreach( 
    t In src.outnbrs 
  ){
    int weight=hgraph.getWeight(src,t);
    if( useLimit && weight>glimit ){
      struct workitem tt;
      tt.src=src; tt.dst=t; tt.weight=weight; tt.cur=cur;
      pending.add(tt);
      return;
    }
    Point (hgraph) rep=hset.find(src);
    Point (hgraph) dst;
    int old,ch;
    if( rep<hgraph.npoints && hgraph.points[rep].minedge==req.weight ){
      Point (hgraph) dst=req.dst;
      int cur=req.cur;
      if( rep!=hgraph.find(t) ){
        struct workitem tt;
        tt.src=src;tt.dst=t;tt.weight=weight;tt.cur=cur;
        next.add(tt);
        while(weight < (old=hgraph.points[rep].minedge)){ MIN(rep.minedge,weight,ch);
          } return;
        }
      }
      }
  }
  }

findLightest1(struct workitem req,Graph hgraph,Collection pred [struct workitem],Set hset[Point(hgraph)]) {
  Point (hgraph) src=req.src;
  Point (hgraph) dst=hset.find(src);
  int cur=req.cur;
  if( rep<hgraph.npoints && hgraph.points[rep].minedge==req.weight ){
    Point (hgraph) dst=req.dst;
    if( dst<hgraph.npoints && (rep=hset.Union(rep,dst)) ){
      hgraph.points[rep].minedge=1234567890;
      struct node tt;
      tt.src=src;
      tt.dst=dst;
      tt.weight=req1.weight;
      mst.add(tt);
    }
    }
  }

Fig. 53: Worklist based MST code for CPU in Falcon part 1
findLightest2(Point p, Graph hgraph, Collection pred [struct workitem], Collection next[struct workitem], Collection pending[struct workitem], int useLimit, Set hset[Point(hgraph)]) {
    struct workitem req;
    req.src = p;
    req.cur = 0;
    findLightest(req, hgraph, pred, next, pending, useLimit, hset);
}

int main(int argc, char *argv[]) {
    hgraph.addPointProperty(minedge, int);
    int a;
    hgraph.read(argv[3]);
    Set hset[Point(hgraph)];
    for(int i = 0; i < hgraph.npoints; i++) hgraph.sortEdges(i);
    glimit = 2000;
    current = &WL1;
    next = &WL2;
    pending = &WL3;
    int bcnt = 0;
    for(int i = 0; i < hgraph.npoints; i++)
        hgraph.points[i].minedge = 1234567890;
    foreach(t In hgraph.points) findLightest2(t, hgraph, *current, *next, *pending, bcnt, hset);
    bcnt = 1;
    while(1) {
        while(1) {
            foreach(t In *current)
                findLightest1(t, hgraph, *current, hset);
            temp = current; current = next; next = temp;
            foreach(t In *current)
                findLightest(t, hgraph, *current, *next, *pending, bcnt, hset);
            if(*next.empty()) break;
        }
        temp = current; current = pending; next = temp;
        if(*pending.empty()) break;
    }
}

Fig. 54: Worklist bade MST code for CPU in Falcon part2