Custom Code Generation for a Graph DSL

(Dual Degree Project - Interim Report)

Bikash Gogoi (CS14B039) *

November 12, 2018

Abstract

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

*Guide : Dr. Rupesh Nasre (Dept of Computer Science & Engineering, IIT Madras)

Contents

1 Introduction

1.1 Motivation

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

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

It would be ideal if the algorithm is programmed at a high-level without worrying about the nuances of the hardware. This gave rise to domain-specific languages (DSLs) for graph algorithms which allowed programmers to write complex algorithmic codes and took care of efficient parallel code generation [11, 34, 2]. Such DSLs often disallow writing arbitrary programs, trading off generality for performance. This makes programming parallel hardware easy, and adapting to changes manageable.

An example of a graph DSL is Falcon [2, 1] which supports a wide variety of backends: CPU, GPU, multi-GPU, heterogeneous processing, and distributed systems. It extends C language to allow graph processing being specified at a high-level. Falcon provides special constructs (such as worklists and reductions) for simplifying algorithm specification and also aiding efficient code generation. We choose Falcon because it supports a variety of backends.

Unfortunately, graph algorithms do not always have a fixed performance pattern. The pattern varies depending upon the graph structure which is available only at runtime. Such an *irregular* pattern poses challenges in parallelizing graph algorithms and in optimizing them for efficient execution. For instance, vertex-based processing works well for road networks, but social networks demand an edge-based processing. Sequential processing of parallel loops demands synchronous execution, but independent loops can be more efficient with asynchronous processing. Similarly, backend optimizations are quite different for different targets such as CPU and GPU.

Falcon (and other graph DSLs) allows writing code for a particular kind of processing. The code written in Falcon DSL needs modifications for an alternative way of processing. Various syntactic elements in the program need to be changed for the alternative way. Thus, code needs to be written separately for vertex-based and edge-based processing, for instance. While this is a step better than being totally rigid, it would be helpful if one could generate different kind of code from the *same DSL specification*. Such a setup greatly simplifies algorithmic specification, and also allows generating code for various situations / backends / graph types from the same specification.

The burden, in this setup, shifts from DSL programmer to the compiler. The only input that the programmer needs to specify (apart from the DSL code) is the type of processing required. This can be easily specified via a command-line switch (e.g., `-vertex-based`, `-synchronous`, etc.), but without needing any modifications to the code. This allows our compiler to generate vertex-based and edge-based graph processing code from the same algorithmic specification. Similarly, it allows generating synchronous and asynchronous processing code, wherein various iterations of the parallel processing are separated and not separated by a barrier respectively. The compiler is also equipped to generate CPU or GPU or multi-GPU code.

1.2 Our Contribution

In this report, we highlight the challenges faced in keeping the specification fixed. In particular, we make the following contributions:

- We present a compiler which generates different implementations for the *same DSL program* for an

algorithm which differ in graph processing. In particular, the compiler can generate vertex-based or edge-based processing, synchronous or asynchronous codes, and CPU or GPU or multi-GPU codes.

- The DSL is able to manage multiple GPUs present in a multi-GPU machine by adding support for running programs in parallel which differ in algorithm or input graph.
- We illustrate the effectiveness of the proposed compiler using several graph algorithms and several graphs of various types. The performance of the code generated with the proposed compiler is compared against other hand-tuned as well as generated codes.

1.3 Outline

The rest of the report is organized as below. Chapter 2 provides a brief background of Falcon DSL, as our work adds facilities in it. Chapter 3 presents challenges faced in generating various codes from the same DSL. Chapter 4 evaluates the effectiveness of our approach using multiple frameworks, algorithms and graphs. Chapter 5 compares and contrasts the relevant related work, and Chapter 6 concludes.

Tool	DSL	Hardware Support CPU: GPU: multi-GPU	Iterators Edge : Vertex : Worklist
GreenMarl	√	√ : × : ×	√ : √ : √
Galois	×	√ : × : ×	× : × : √
Falcon	√	√ : √ : √	√ : √ : √
Totem	×	√ : √ : √	× : × : ×
Gunrock	×	× : √ : ×	× : × : √
LonestarGPU	×	× : √ : ×	√ : √ : √

Table 1: Comparison of different graph frameworks

2 Background

We provide a comparison of a few graph frameworks, followed by a brief background on Falcon.

2.1 Comparison of Different Graph Frameworks

Table 1 differentiates various graph data analytic tools based on programming style, hardware support and processing style.

Galois [28] is a framework for graph algorithms. It iterates over a collection of active elements which are stored in a worklist. After one parallel iteration over a worklist, new active elements are generated which are stored in another worklist for next iteration. Program terminates when no new active elements are generated during an iteration. Galois supports different types of worklists. It supports dynamic algorithms, where graph structure changes at runtime. Green-Marl [12] is a Graph DSL. The DSL supports various data types, parallel constructs to iterate over graph, synchronization and reduction functions. This makes programming graph algorithms using Green-Marl DSL easy. Galois and Green-Marl target multi-core CPUs and do not support `GPU` devices.

LonestarGPU [26] framework supports dynamic algorithms. It has implementations of dynamic graph algorithms such as Delaunay Mesh Refinement and Survey Propagation. Static graph algorithms such as SSSP, BFS, Connected Components etc., are also programmed in LonestarGPU. Being a framework (and not a DSL), programming new algorithm in LonestarGPU require expertise in CUDA and framework code. Gunrock [37] has a set of APIs to express a wide range of graph processing primitives on GPUs. It defines *frontiers* as a subset of edges and vertices of the graph which are actively involved in the computation. Gunrock defines advance, filter, and compute primitives which operate on frontiers in different ways. Similar to LonestarGPU, Gunrock is a library-based framework and, relative to DSLs, programming is difficult.

Falcon [2] is a Graph DSL which supports `CPU`, `GPU` and multi-GPU machines. It supports various data types, parallelization and synchronization constructs, and reduction operations. This makes programming graph analytic algorithms easy for heterogeneous targets. Falcon also supports dynamic graph algorithms. Totem [8] is a library-based framework consisting of benchmarks which can run on `CPU`, `GPU` and multi-GPU machines. Programming new algorithms in Totem is relatively tougher.

As shown in Table 1, both Green-Marl and Falcon support iteration over nodes and edges. Falcon supports multiple hardware platforms. So we have taken Falcon DSL for implementing our compilation strategies where several optimizations are performed. These optimizations are missing in any of the current graph DSL. By retaining the DSL code intact for various transformations, our compiler considerably reduces the programming effort and improves productivity.

2.2 Falcon

Falcon Graph DSL has data types `Graph`, `Point`, `Edge`, `Set` and `Collection`. `Graph` stores a graph object, which consist of points and edges. Each `Point` is stored as `union` of `int` and `float`. `Edge` consist of source and destination points and a *weight*. `Set` is a static collection and implemented as a *Union-Find*

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

Table 2: `foreach` statement iterators in Falcon

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

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

data structure. The `Collection` data type is dynamic and its size can vary at runtime. Elements can be added to and deleted from a collection object at runtime.

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

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

Both the algorithms are for Single Source Shortest Path (SSSP) computation. It computes shortest path from source point (point zero) to all other points in the graph object. In Algorithm 1, the processing is done using *points* (Line 15) and *outnbrs* (Line 4) iterators. In Algorithm 2, the computation is performed using *edges* (Line 17) iterator. In both the algorithms all the edges $t \rightarrow p$ in the graph object are considered. Then *dist* value of point t is reduced to $Min(t.dist, p.dist + weight(p \rightarrow t))$ using the atomic function *MIN*. If there is any change in the value of *t.dist*, the variable *changed* is set to one. The computation stops when the value of *dist* does not change for any point in the graph object. Performance of an algorithm depends on the graph structure, hardware architecture, etc. Algorithm 1 may perform well over Algorithm 2 for one input graph, but may not for another, on the same hardware architecture. This depends upon several graph properties such as variance in degree, diameter of graph, etc.

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

The random graphs (Erdös R*ényi* model) typically perform well with iterating over *points*. The social and rmat graphs which follow power-law degree distribution [8] are benefited mostly by iterating over *edges*, especially on `GPU` devices. Power-law degree distribution indicates huge variance in degree distribution of the vertices. This can result in thread-divergence in `GPU`, when parallelized over *points* and iterated over

Algorithm 1: SSSP: iterating over Points in Falcon

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

Algorithm 2: SSSP: iterating over Edges in Falcon

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

their *outnbrs* or *innbrs*.

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

3 Our Approach

Algorithm 3 shows the code fragment of the SSSP computation in Algorithm 1. Only parts which are relevant for our compiler transformation are shown here. The algorithm is vertex-based SSSP computation using *points* and *outnbrs* iterators. If the target device is specified as `CPU`, then parallel C++ code with `OpenMP` gets generated. A programmer may request vertex-based code using the relevant command line argument during compilation of the above DSL code. The compilation happens similar to the original Falcon compiler and the output would be the same as that generated by Falcon.

Algorithm 3: SSSP: iterating over Points in Falcon

```
1  int changed = 0; // Global variable
2  relaxgraph(Point p, Graph graph) {
3      foreach (t In p.outnbrs)
4        MIN(t.dist, p.dist + graph.getweight(p, t),
         changed);
5  }
6  main(int argc, char *argv[]) {
7      .....
8      while (1) do
9          changed = 0;
10         foreach (t In graph.points) relaxgraph(t,graph);
11         if (changed == 0) break; //reached fix point
12     end
13 }
```

A programmer may require parallel C++ code which is similar to the parallel C++ SSSP generated by Falcon using *edges* iterator (see Algorithm 2). In Falcon, the programmer has to code it separately. This means, one needs to write a separate code for vertex-based processing, another code for edge-based processing, yet another for asynchronous variants of these, and so on. It would be ideal if the programmer simply specifies *what* rather than *how*, and the DSL compiler takes care of appropriate code generation. We support it in this work. In our proposal, the programmer simply needs to specify different compile-time arguments. This triggers generation of parallel C++ code matching the output of the Falcon compiler for Algorithm 2. Thus, coding efforts considerably reduce.

Unnikrishnan et al. [2] discuss how Falcon converts a DSL code matching the iterators present in the DSL code. Here, we explain our compiler transformation. That is *points+outnbrs* based DSL code generates parallel C++ code which matches the output of *edges* iterator based Falcon DSL code. The `foreach` calls at Lines 10 and 3 make all the edges of Algorithm 3 to get processed in the code enclosed inside the *outnbrs* iterators. Our compiler transformation removes the `foreach` in relaxgraph function (Line 3 and converts the `foreach` in Line 10 to edges iterator. The first argument to *relaxgraph()* is modified to an `Edge` object from a `Point` object. In Algorithm 3 the iterator instances with name p and t act as source and destination vertices of the edge $p \rightarrow t$ of the graph. Thus, in the relaxgraph function, we instantiate p and t using source and destination vertices of the edge. Note that our transformation has modified the first argument to relaxgraph as of type edge now. This is followed by the Falcon compiler generating the parallel C++/CUDA code. This transformation requires modification of Falcon Abstract Syntax tree (AST). We now highlight the challenges faced in generating code from the same DSL code.

3.1 Vertex-based versus Edge-based

Conversion of vertex-based to edge-based and vice-versa are done completely at the abstract-syntax tree (AST) level by traversing the AST and modifying its eligible parts. An important conversion non-triviality stems from the edge-based processing being a single loop, while the corresponding vertex-based processing is a nested loop (outer loop over vertices, and inner loop over all the neighbors of each vertex). Pseudo-code for the transformation is presented in Algorithm 4. Step-1 in the algorithm marks the outer *foreach* statements. Step-2 transforms vertex-based code to edge-based code. Step-3 transforms edge-based code to vertex-based code.

In vertex-based to edge-based conversion, the subtree is eligible for conversion if the following two conditions are met. First, the subtree is rooted at a function node whose only child is a node for a `foreach` which iterates through a point's neighbors. Second, the function is the only statement in the body of a `foreach` which iterates through the graph-points. Once the eligible parts are found, we switch the points iterator of the `foreach` statement from which the function is called to edge iterator, and then remove the

`foreach` statement in the function. The conversion also requires change in the function's signature as its argument was earlier a point, while now it is an edge. It also necessitates defining two new variables at the beginning of the function corresponding to the source and the destination of the edge. The name of one of the two variables is the name of the point which was the former parameter of the function. The other variable's name is derived from the iterator of the `foreach` statement removed from the function earlier. The order in which these names are mapped to the variables depends on the iterator used in the removed `foreach`. If the iterator is over out-neighbors, the name of the iterator is mapped to the destination vertex of the edge. Otherwise, we map it to the source vertex. Such an implementation allows the rest of the processing in the iteration to be arbitrary, and reduces the number of alterations the compiler needs to perform to the code.

In edge-based to vertex-based conversion, the compiler needs to do the opposite (see Algorithm 4). Here, it finds the `foreach` statement iterating through the edges of a graph which contains a function call statement as the only statement in the loop-body. For such a `foreach`, the iterator is changed to iterate over points and a new `foreach` statement iterating through the neighbors of the point is introduced in the called function (which encloses the body of the called function). This conversion necessitates the parameter of the called function to be changed from edge type to point type. It also requires defining a new variable which represents the edge between the point passed as a parameter and the iterator which represents the point's neighbor. Such an implementation also allows the rest of the processing in the iteration to be arbitrary.

Algorithm 4: Code Transformation

```
   Input: Falcon vertex/edge based DSL code
   Output: C++/CUDA edge or vertex based code based on target
 1  begin Step1:- Mark outermost foreach statement (Done in Falcon Parser).
 2  if statement.type==foreach && level==0 then
 3  |   t.outer=true;
 4  end
 5  if statement.type==foreach && level==1 then
 6  |   t.outer=false;
 7  end
 8  end Step1
 9  begin Step2:- Convert vertex code to edge code
10  forall foreach statement f1 in program do
11  |   if f1.outer== true && f1.iterator==points then
12  |   |   forall foreach statement f2 in program do
13  |   |   |   if f2.def.fun == f1.call.fun && f2.itr == outbrs || f2.itr == innbrs then
14  |   |   |   |   //modify iterator of f1 to points
15  |   |   |   |   // modify 1st argument to Edge in f2.def.fun
16  |   |   |   |   //create f2.itr and f1.itr in f2.def.fun using Edge.
17  |   |   |   |   // remove foreach in f2.def.fun
18  |   |   |   |   // generate code (Done by Falcon)
19  |   |   |   end
20  |   |   end
21  |   end
22  end
23  end Step2
24  begin Step3:- Convert edge code to vertex code
25  forall foreach statement f1 in program do
26  |   if f1.outer==true && f1.iterator==edges then
27  |   |   function fun =f1.call.fun
28  |   |   modify iterator of fun to points
29  |   |   // modify 1st argument of fun to Point
30  |   |   // insert outnbrs or innbrs iterator in fun using Point
31  |   |   // generate code (Done by Falcon)
32  |   end
33  end
34  end Step3
```

An important artifact of this processing conversion is that it affects the way graph is stored in memory. In vertex-based code, Falcon (and other frameworks) store the graph in compressed sparse-row (CSR) format. Compared to edge-list representation, CSR format reduces the storage requirement and has the benefit of quick access to the out-neighbors of a vertex. In edge-based codes, on the other hand, graph is stored in edge-list format (source destination weight) which enables quick retrieval of the source and the destination points of an edge. However, one graph format also forbids the other format's advantages. For instance, if the graph is stored in CSR format, retrieving an arbitrary edge (say, edge $p \rightarrow q$) is time-consuming (requires search over the out-neighbors of p). On the other hand, if the graph is stored as edge-list, retrieving an edge, given the source and the destination vertices is even more time-consuming, as the edges are not indexed on vertices. In many cases (and all in our testbed), accessing edges is not arbitrary. Edges are accessed in a particular order, for instance, iterating through out-neighbors of a point. So edge between a point and its out-neighbor can be retrieved quickly if the graph is stored in CSR format. Unfortunately, while changing vertex-based code to edge-based code, accessing edges in this manner becomes time-consuming. To overcome this, the code where the edge is retrieved is replaced by

the edge passed as an argument to the transformed function. This resolves the issue of enumerating over neighbors.

3.2 Synchronous versus Asynchronous

By default, Falcon generates synchronous code, that is, it inserts a barrier at the end of parallel construct. While this works well in several codes and eases arguing about the correctness (due to data-races restricted to within-iteration processing across threads), synchronous processing may be overly prohibitive in certain contexts. Especially, in cases where processing across iterations is independent and the hardware does not necessarily demand single-instruction multiple data (SIMD) execution, asynchronous processing may improve performance. Arguing about the correctness-guarantees gets so involved in asynchronous execution, that some DSLs strictly enforce synchronous code generation only. Our proposal is to allow the programmer to generate synchronous or asynchronous code without having to change the algorithm specification code in the DSL. Achieving this necessitates identifying independent processing in the code. Towards this, we maintain read and write sets of global variables and the graph attributes used in each *target function* separately. A *target function* is a function which is being called in the body of a `foreach` statement and the function call is the only statement inside the body of the `foreach`. On CPU, it is the parallel loop body, while on GPU, this function becomes the kernel.

This is a two-step process. In Step 1, we mark nodes in the control-flow graph (CFG); and in Step 2, we generate the appropriate code. In Step 1, we construct the CFG of the *target function* call. Using the read and the write sets corresponding to each of the *target function*s, we mark each node of the CFG as *barrier-free* or not. A barrier-free node signifies that the *target function* corresponding to the barrier-free node can be executed concurrently with the children of this node. A node is barrier-free if it satisfies the following two conditions:

1. There is no dependency (Read-Write, Write-Read and Write-Write) between the node and each of its children.

2. There is no dependency between the node and the codes between the node and its children.

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

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

Discussion: The way we group two parallel constructs may not lead to optimal grouping. Finding the optimal grouping is not feasible as it depends on the time required to execute the parallel constructs. For instance, A, B, and C are parallel constructs where A and B are independent, B and C are independent, and A and C are dependent. If time required to execute A is less compared to B and C, then grouping A and B

does not lead to optimal performance. On the other hand, if the time required to execute C is less compared to A and B, then grouping A and B does lead to optimal performance.

3.3 CPU, GPU and Multi-GPU Codes

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

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

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

3.4 Data-Transfer Aggregation Optimization

In Falcon, custom property of points/edges in a graph is stored as an array where n^{th} position stores the value corresponding to point/edge n. If a property of a point/edge of a GPU graph is read in CPU, a data transfer from GPU to CPU is necessary. For n such reads, n instances of data transfer are required. This is not always desirable and can quickly become a performance bottleneck due to slow PCI-e across the devices. To overcome this, we combine multiple such data transfers into a single data transfer, where feasible.

Such an optimization is particularly beneficial for *for* and *while* loops. If the instructions inside the body of a *for/while* loop only read a particular property (and may write to other properties), we copy the array related to that property to CPU before the beginning of the loop using a single data transfer instruction. The instructions in the loop then use the CPU copy. This often improves the performance as a loop is expected to be executed several times.

Graph	#nodes $\times 10^6$	#edges $\times 10^6$	max-degree
USA-road-CTR	14	34	9
USA-road-USA	24	58	9
soc-orkut-dir	3	234	33313
soc-sinaweibo	21	261	278491
random-25M	25	100	17
random-50M	50	200	18
random-75M	75	300	18
random-100M	100	400	18
random-125M	125	500	19
rmat-10M	10	100	1873
rmat-20M	20	200	2525
rmat-30M	30	300	3796
rmat-40M	40	400	3333
rmat-50M	50	500	4132

Table 4: Benchmark characteristics

Graph	BFS	CC	MST	SSSP
USA-road-CTR	3456	14103	727	30299
USA-road-USA	9113	24061	779	72857
soc-orkut-dir	269	623	5509	267
soc-sinaweibo	1234	1955	30556	1151
random-25M	131	411	2832	561
random-50M	270	823	6665	1142
random-75M	414	1416	11265	1796
random-100M	583	2192	15860	2413
random-125M	756	2909	25973	3194
rmat-10M	133	387	2770	536
rmat-20M	266	789	6240	1169
rmat-30M	424	893	8963	1859
rmat-40M	542	1601	13232	2405
rmat-50M	707	2026	16825	3808
Total	18298	54189	148196	123457

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

4 Experimental Evaluation

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

4.1 Experimental Setup

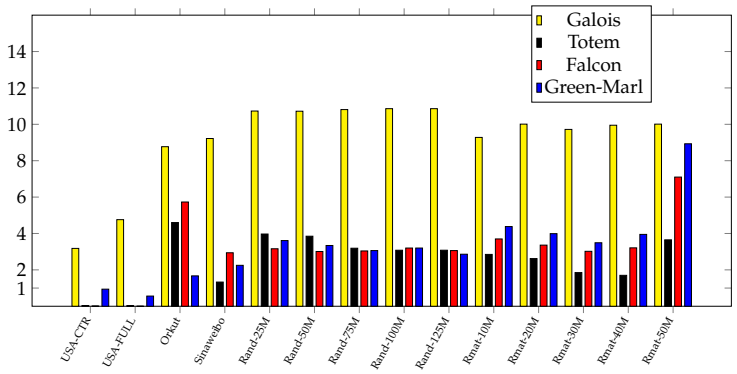
We used a range of graph types to assess the effectiveness of our proposal. The dataset graphs in our experimental setup and their characteristics are presented in Table 4. We used four graph algorithms in our testbed: Breadth-First Search (BFS), Connected Components (CC), Minimum Spanning Tree computation (MST) and Single-Source Shortest Paths computation (SSSP). All these algorithms are fundamental in graph theory and form building blocks in various application domain. We compare the generated codes against the following frameworks: Galois [28], Totem [8], Green-Marl [12], LonestarGPU [26] and Gunrock [37]. In the sequel, Falcon refers to our proposed techniques embedded into existing Falcon.

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

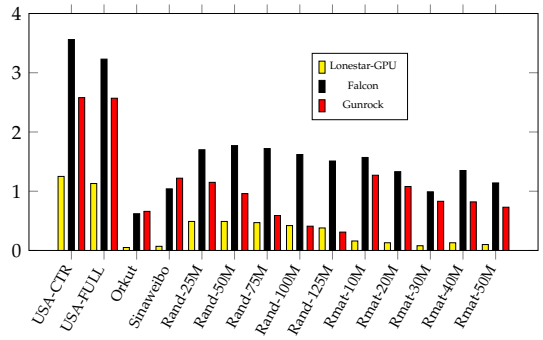
4.2 Baselines and Comparison with Other Frameworks

The baseline execution times of Falcon on GPU are listed in Table 5. We observe that the execution times on road networks are particularly high for propagation based algorithms such as BFS, SSSP and CC. This occurs because unlike other graphs, road networks have large diameters, leading to many iterations of the algorithm.

Figure 1, Figure 2 and Figrue 3 compares the performance of our modified Falcon against other frameworks. For SSSP in GPU, we have observed that Falcon-generated code provides consistently better speedups compared to LonestarGPU and Gunrock, except on the two social networks (soc-orkut-dir and soc-sinaweibo). Totem performs better on the social networks as well as on RMAT graphs due to its inbuilt edge-based processing and other optimizations to improve load-balancing across GPU threads. In CPU version of SSSP, we have observed that Galois performs better than all other frameworks. Performance of Falcon, Totem and Green-Marl are similar. For BFS in GPU, the results are mixed across various frameworks

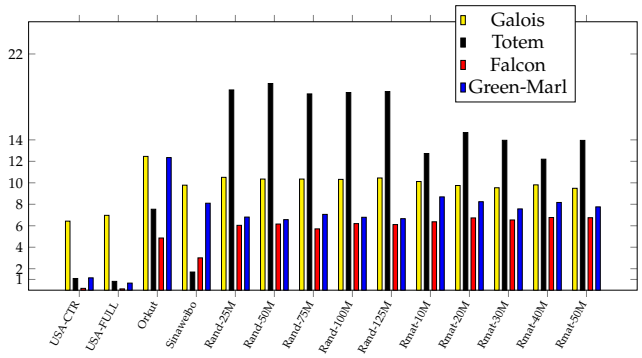


(a) CPU: Speedup over Galois single thread

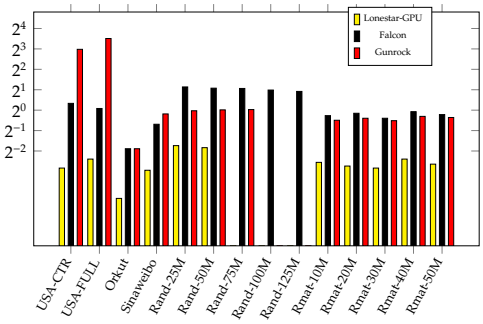


(b) GPU: Speedup over Totem

Figure 1: SSSP comparison

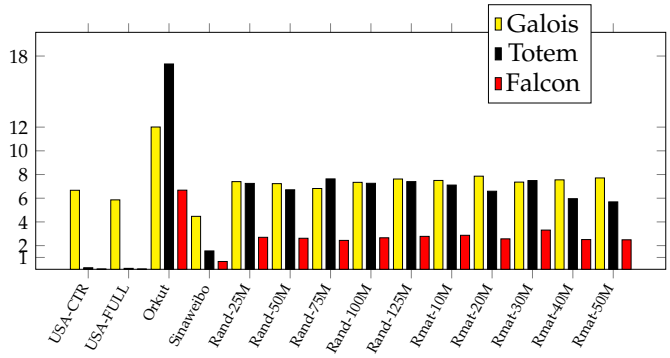


(a) CPU: Speedup Over Galois single thread

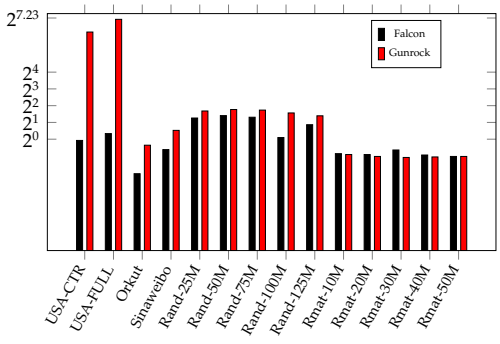


(b) GPU: Speedup over Totem

Figure 2: BFS comparison

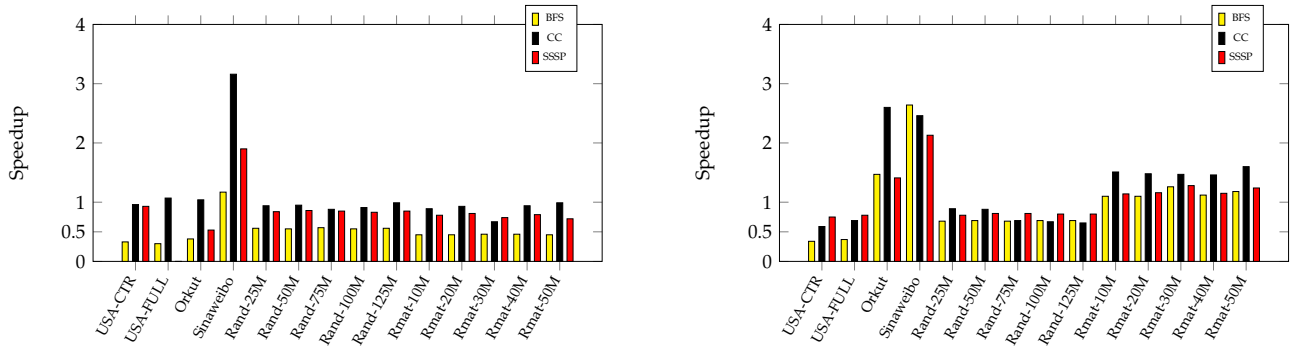


(a) CPU: Speedup over Galois single thread



(b) GPU: Speedup over Totem

Figure 3: CC comparison



(a) CPU (b) GPU

Figure 4: Speedup of edge-based over vertex-based processing

and there is no clear winner, but there are interesting patterns based on the graph types. Gunrock performs quite well on the road networks (USA-road-USA and USA-road-CTR), primarily due to its work-efficient worklist-based processing. Totem outperforms again on social networks due to edge-based processing and better load-balancing. Performance of almost all the frameworks on RMAT graphs is quite similar, with LonestarGPU performing poorly. Our Falcon stands out on random graphs with speedups close to 2× over all other frameworks. In CPU, Totem is a clear winner for random and RMAT graphs whereas Galois performed better in road and social networks. For GPU version of CC, Falcon performed better in road network, social network and random graphs. For RMAT graph, all the frameworks are similar. In CPU version, performance of Galois and Totem are similar and better than Falcon.

4.3 Effect of Vertex-based versus Edge-based

Figure 4a presents results of edge-based versus vertex-based processing of Falcon across various graphs for CC, BFS and SSSP. We observe that edge-based processing performs better in social-networks (soc-orkut-dir and soc-sinaweibo) and RMAT graphs. Both these kinds of graphs have skewed (power-law) degree-distribution resulting in large load-imbalance with vertex-based processing. These graphs follow small-world property due to this peculiar (and natural) degree distribution. On GPUs, this load-imbalance manifests itself as thread-divergence as the number of iterations (based on the number of neighbors) of each thread has high variance. In other words, threads mapped to vertices having few neighbors have to wait for others mapped to high-degree vertices. This inhibits parallelism for SIMD style of processing. In contrast, in edge-based processing, since threads are mapped to (a group of) edges, the load-imbalance is relatively negligible. This results in better thread-divergence among warp-threads, leading to improved execution time. Road networks and random graphs, on the other hand, have quite uniform degree-distribution. Therefore, edge-based processing is not very helpful. In fact, for uniform degree-distributions, edge-based processing may lead to inferior results (as seen in our experiments), due to increased synchronization requirement. Different outgoing edges of a vertex are processed sequentially by the same thread in vertex-based processing; whereas, those are processed in parallel by different threads. Thus, edge-based processing necessitates more coordination among threads with respect to reading and updating attribute values of vertices.

Figure 4b presents results of edge-based processing of Falcon on CPU. We observe that, unlike on GPUs, edge-based processing is not helpful on GPUs. This is primarily due to CPUs not having enough resources to utilize the additional parallelism exposed by edge-based processing. Thus, a few tens of threads perform in a similar manner in the presence of a million vertices or multi-million edges. The only exception is soc-sinaweibo graph which witnesses over 3× speedup for CC on CPU due to edge-based processing. The improvements on this graph are also high for other algorithms as well (BFS and SSSP) compared to

Graph	Synchronous	Asynchronous
USA-road-CTR	86	91
USA-road-USA	134	106
soc-orkut-dir	425	285
soc-sinaweibo	8310	4508
random-25M	458	347
random-50M	945	741
rmat-10M	329	320
rmat-20M	771	665
Average	1433	882

Table 6: Execution times (in ms) for Synchronous versus Asynchronous processing in CPU

Graphs	CPU-16	GPU	multi-GPU
random-25M + random-50M	3866	1234	826
rmat-10M + rmat-20M	3024	1176	792

Table 7: Execution time (in ms) of CC for two different graphs for various targets

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

Table 8: Execution time (in ms) of BFS for two different graphs for various targets

other graphs. This occurs due to higher average degree in this social network (see Table 4).Higher average degree adds sequentiality in vertex-based processing, while edge-based processing is not amenable to degree-distribution or average degree. The overall effect gets pronounced for such dense graphs.

4.4 Effect of Synchronous versus Asynchronous Processing

Table 6 presents the effect of asynchronous processing for various graphs on CPU. We used a combination of BFS and SSSP to perform independent processing on the same graph. We observe that asynchronous version improves execution time by 38%. This occurs because threads do not have to wait for other threads. This is primarily true on CPUs as threads are monolithically working on different parts of the graph and seldom require synchronization. Such a processing is likely to benefit performance on GPUs as well, but due to limited GPU resources, the asynchronous kernels could not be executed together. Therefore, our observed performance was the same for synchronous and asynchronous processing on GPUs (hence not shown again).

4.5 Effect of Code Generation for Multiple Targets

As discussed in Section 3.3, our approach can seamlessly generate code for CPU or GPU or multi-GPU. The multi-GPU code works with different graphs for the same algorithm, or with the same graph for different algorithms. Table 7 presents results for the former with CC as the algorithm generating code for CPU with 16 threads, single GPU and two GPUs, while Table 8 presents those for BFS. We observe that multi-GPU version took much less time as compared to other backends. In both CPU and single-GPU versions, the graphs are processed one after another. On the other hand, in multi-GPU version, both the graphs are processed simultaneously in different GPUs; so the overall execution time is the larger of the two.

4.6 Effect of Data-Transfer Optimization

To illustrate the effect of the memcpy-optimization, we devised a simple Falcon program which computes BFS on GPU and then queries distances of various vertices from the CPU to compute the maximum distance. We observe that the program with optimization takes less than a second to find the maximum distance. On the other hand, without optimization the same program takes more than five minutes. This happens

because without optimization, the existing Falcon engine generates code to copy distance of each vertex one by one as required in each iteration (total N small `cudaMemcpy`s). In contrast, the optimized code copies all the distances at once and then uses it for finding maximum distance (one large `cudaMemcpy`). This leads to considerably reduced communication overhead, leading to improved execution.

5 Related Work

We compare with the relevant literature, by dividing it based on the type of parallel processing: multi-core, GPUs-based, and distributed.

5.1 Multi-core CPU

Green-Marl [12] is a graph DSL for implementing parallel graph algorithms on multi-core CPUs using `OpenMP`. It has two data types `Node` and `Edge` to represent vertices and edges in the graph respectively. Green-Marl does not support mutation of the graph object (i.e., adding and removing vertices and edges to/from the graph object). So dynamic graph algorithms cannot be written in Green-Marl. LightHouse [34] added CUDA backend to Green-Marl. Galois [28] is a C++ framework for implementing graph algorithms on multi-core CPUs. It supports mutation of graph objects via *cautious* speculative execution. It uses a *worklist* based execution model, where all the *active elements* are pushed to a *worklist* and are processed in *ordered* or *unordered* fashion. Elixir [29] is a graph DSL to develop and implement parallel graph algorithms for analyzing static (i.e., non-mutable) graphs and it targets multi-core CPUs.

X-Stream [31] uses edge-centric processing for graph applications. . It supports both in-memory and out-of-core graph processing on a single shared-memory machine using scatter-gather execution model. The Stanford Network Analysis Platform (SNAP) [20] provides high-level operations for large network analysis including social networks and target multi-core CPUs. Ligra [35] is a framework for writing graph traversal algorithms for multi-core shared memory systems. For vertex- versus edge-based processing, it uses two different routines: one for mapping vertices and the other for mapping edges. However, the DSL code needs modification if one needs to alter the mapping. Polymer [40] is a NUMA aware graph framework for multi-core CPUs and it is built with a hierarchical barrier to get more parallelism and locality. The GraphChi [18] framework processes large graphs using a single machine, with the graph being split into parts, called shards, and loading shards one by one into RAM and then processing each shard. Such a framework is useful in the absence of distributed clusters.

5.2 `GPU` devices

Due to *irregularity* present in the graph algorithms thread divergence can happen in `GPU`. Past research has shown that graph algorithms perform well on GPUs and much better than multi-core `CPU` even in the presence of the above mentioned limitations. The BFS implementation from Merrill [24] is novel and efficient. There have also been successful implementations of other local computation algorithms such as betweenness centrality [33] and data flow analysis [23] on `GPU`. Different ways of writing SSSP programs (such as delta-stepping [25]) on `GPU` along with their merits and demerits have been explored in [5] and it concludes that worklist-based implementation will not benefit much on `GPU` compared to that on a `CPU`.

The Lonestar-GPU [26] framework supports mutation of graph objects and implementation of cautious morph algorithms on `GPU`. It has cautious morph implementations of algorithms like Delaunay Mesh Refinement, Survey Propagation and Points-to-Analysis. Medusa [41] is a programming framework for graph algorithms on GPUs and multi-GPU devices. It provides a set of APIs and a run time system to program graph algorithms targeting GPU devices. The programmer is required to write only sequential C++ code with these APIs. APIs on vertices and edges can also send messages to neighboring vertices. The Gunrock [37] framework provides a data-centric abstraction for graph operations at a higher level which makes programming graph algorithms easy. Gunrock has a set of APIs to express a wide range of graph processing primitives. Gunrock also has some GPU-specific optimizations. In Gunrock, programs can be specified as a series of bulk-synchronous steps. Totem [8, 9] is a heterogeneous framework for graph processing on a single machine. It supports using a multi-core CPU and multiple GPUs on a single machine. Totem follows the Bulk Synchronous Parallel [36] model of execution. IrGL [27] implements three optimizations named *iteration outlining*, *cooperative conversion* and parallel execution of nested loops. IrGL is an intermediate code representation, on which these optimizations are applied and the CUDA

code is generated from it. Farzad et al. [16] propose warp segmentation to improve GPU utilization by dynamically assigning appropriate number of threads to process a vertex. GasCL (Gather-Apply-Scatter with OpenCL) [6] is a graph processing framework built on top of OpenCL which works on several accelerators and supports parallel work distribution and message passing. The MapGraph [7] framework provides high-level APIs, making it easy to write graph programs and obtain good speedups on GPUs. Halide [30] is a programming model for image processing on CPUs and GPUs. An online profiling based method [14] partitions work and distributes it across CPU and GPU. CuSha [17] proposes two new ways of storing graphs on GPU called G-Shards and Concatenated Windows, that have improved regular memory access patterns. OpenMP to GPGPU [19] is a framework for automatic code generation for GPU from OpenMP CPU code. There is no support from the CUDA compiler to have a barrier for the all threads in a *kernel* blocks. Such a feature is needed in some *cautious* morph algorithm (e.g, DMR). A barrier for all the threads in a *kernel* can be implemented in software, by launching the *kernel* with less number of threads and with the help of *atomic* operations provided by CUDA, and each thread processes a set of elements. Such an implementation can be found in [39].

5.3 Distributed Systems

Natural graphs have very big sizes. Such large-scale graphs are sparse and follow the power-law degree distribution. Such graphs are processed on a computer cluster. Programming for a computer cluster requires learning the MPI library and explicit communication code has to be inserted in the program, with proper synchronizations to preserve sequential consistency. To achieve good performance there should be work balance across machines in the cluster and communication overhead should be minimum. Also the graph should be partitioned across machines with less storage overhead. GraphLab [21], PowerGraph [10] and Pregel [22] are popular distributed graph analytics framework. Bulk Synchronous Parallel (BSP) Model [36] of execution and asynchronous executions are popular models of executions. Giraph [3] is an open source framework written in *Java* which is based on the Pregel model and runs on the Hadoop infrastructure. GPS (Graph Processing System) [32] is an open source framework and follows the execution of model of Pregel. Green-Marl compiler was extended to CPU-clusters [13] and it generates GPS based Pregel like code. Mizan [15] uses dynamic monitoring of algorithm execution, irrespective of graph input and does vertex migration at run time to balance computation and communication. Hadoop [38] follows the MapReduce() processing of graphs and uses the Hadoop distributed file system (HDFS) for storing data. Pregel like systems can outperform MapReduce() systems in graph analytic applications. Gluon [4] uses Galois and Ligra and generates distributed-memory versions of these systems. Gemini [42] is a distributed graph processing framework and provides abstractions for push-pull model of computation on distributed systems.

6 Conclusion

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

References

[1] U. Cheramangalath, R. Nasre, and Y. N. Srikant. DH-Falcon: A Language for Large-Scale Graph Processing on Distributed Heterogeneous Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 439–450, USA, Sept 2017. IEEE.

[2] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 12(4):54:1–54:27, December 2015.

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[4] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 752–768, New York, NY, USA, 2018. ACM.

[5] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, Washington, DC, USA, 2014. IEEE Computer Society.

[6] E. Elsen and V. Vaidyanathan. A vertex-centric cuda/c++ api for large graph analytics on gpus using the gather-apply-scatter abstraction, 2013.

[7] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[8] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM.

[9] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. The Energy Case for Graph Processing on Hybrid CPU and GPU Systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[10] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[11] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[13] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 208:208–208:218, New York, NY, USA, 2014. ACM.

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

[15] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[16] Farzad Khorasani. *High Performance Vertex-Centric Graph Analytics on GPUs*. PhD thesis, University of California Riverside, 2016.

[17] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[18] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[19] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009.

[20] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, July 2016.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[23] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[24] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[25] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003.

[26] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. *SIGPLAN Not.*, 48(8):147–156, February 2013.

[27] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[28] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[29] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A System for Synthesizing Concurrent Graph Programs. *SIGPLAN Not.*, 47(10):375–394, October 2012.

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

[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[32] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[33] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness Centrality on GPUs and Heterogeneous Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, New York, NY, USA, 2013. ACM.

[34] G. Shashidhar and Rupesh Nasre. LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 1–10, USA, 2016. IEEE.

[35] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[36] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8):11:1–11:12, February 2016.

[38] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., USA, 1st edition, 2009.

[39] S. Xiao and W. c. Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, USA, April 2010. IEEE.

[40] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015.

[41] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, June 2014.

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association.