Code Generation for a Variety of Accelerators for a Graph DSL

ASHWINA KUMAR, IIT Madras, India
M. VENKATA KRISHNA, PSG Tech, India
PRASANNA BARTAKKE, IIT Madras, India
RAHUL KUMAR, IIT Madras, India
RAJESH PANDIAN M, IIT Madras, India
NIBEDITA BEHERA , IIT Madras, India
RUPESH NASRE, IIT Madras, India

Sparse graphs are ubiquitous in real and virtual worlds. With the phenomenal growth in semi-structured and unstructured data, sizes of the underlying graphs have witnessed a rapid growth over the years. Analyzing such large structures necessitates parallel processing, which is challenged by the intrinsic irregularity of sparse computation, memory access, and communication. It would be ideal if programmers and domain-experts get to focus only on the sequential computation and a compiler takes care of auto-generating the parallel code. On the other side, there is a variety in the number of target hardware devices, and achieving optimal performance often demands coding in specific languages or frameworks. Our goal in this work is to focus on a graph DSL which allows the domain-experts to write almost-sequential code, and generate parallel code for different accelerators from the same algorithmic specification. In particular, we illustrate code generation from the StarPlat graph DSL for NVIDIA, AMD, and Intel GPUs using CUDA, OpenCL, SYCL, and OpenACC programming languages. Using a suite of ten large representative graphs and four popular algorithms, we present the efficacy of StarPlat's versatile code generator.

1 ACM REFERENCE FORMAT:

Ashwina Kumar, M. VENKATA KRISHNA, Prasanna Bartakke, Rahul Kumar, Rajesh Pandian M, Nibedita Behera, and Rupesh Nasre. 2023. Code Generation for a Variety of Accelerators for a Graph DSL. In . ACM, New York, NY, USA, 17 pages. https: //doi.org/10.1145/

2 INTRODUCTION

Graphs naturally model several real-world phenomena such as social networks, road networks and biological systems. Computer systems today need to deal with huge graphs: Facebook has 2.9 billion monthly active users, US has a rail network of 260,000 km with a transport of 10.3 billion passenger-km, while our brain network has 86 billion neurons. To deal with such a scale, it is natural to employ farms of parallel machines housing powerful accelerators. We continue to witness various architecture designers and vendors proposing new hardware to enable us process huge data fast.

To accelerate our algorithms and applications optimally on a certain hardware, currently, the domain expert (such as a biologist) needs to program in a certain language or a framework geared towards the hardware. For instance, NVIDIA GPUs are tied closely with CUDA, Intel GPUs expect SYCL, while AMD GPUs need Hip for extracting the best performance. This is clealy not ideal. Therefore, we have been moving towards common computing languages such as OpenCL and OpenACC, which are supposed to be platform portable. Unfortunately, their support is currently limited, and sometimes driven by vendor competition.

In this work, we focus on graph algorithms and accelerators, and help domain-experts generate parallel code for multiple accelerators from the same algorithmic specification. In this endeavour, we employ a recently proposed domain-specific language for graph algorithms named StarPlat [2] and augment its compiler to support multiple accelerator backends: CUDA, SYCL, OpenCL and OpenACC.

This paper makes the following contributions.

- A code generator which builds upon the intermediate representation (abstract syntax tree) of StarPlat to support NVIDIA, Intel, and AMD GPUs. This relieves the domain-expert from learning new languages to write parallel code for graph algorithms.
- Custom processing per accelerator for optimized computation. This allows StarPlat to, for instance, optimize data-clause around loops in OpenACC, while optimize a reduction operation for the SYCL backend.
- An extensive experimental evaluation of the generated codes against library-based Gunrock and manually-optimized LonestarGPU on ten large graphs and four popular graph algorithms (betweenness centrality, page rank, single-source shortest paths, and triangle counting), illustrating StarPlat's ability to be competitive with library-based and hand-crafted codes.

The rest of the paper is organized as follows: Section 3 presents the language specification and various accelerator backends along with an example DSL program. Section 4 describes the code-generation scheme followed for the translation of the DSL code for each backend accelerator. Section 5 provides an overview of the backend-specific optimizations employed for efficient code generation. Section 6 presents the experimental evaluation of the generated code for each backend. Section 7 discusses the related work for graph analytics. We summarise our experience and conclude in Section 8.

3 BACKGROUND

Figure 1 presents the algorithmic specification for computing betweenness centrality (BC) in the StarPlat DSL. It employs Brandes algorithm [3] resembling unweighted all pairs shortest paths (APSP). To be practically tractable, literature usually runs a few iterations of APSP, which can be specified in StarPlat as sourceSet to the ComputeBC function. For each such source (Line 4), the algorithm performs a forward and a backward pass accumulating sigma (number of shortest paths) in the forward pass and updating delta values (fraction of the shortest paths) in the backward pass.

The function takes three parameters: the underlying graph, BC values to be updated, and the sources from where the shortest path computation is to be initiated. Line 2 uses attachNodeProperty which initializes the BC value of each node to 0. The for loop at Line 4 iterates through all the sources in sourceSet. Lines 5 and 6 define new node attributes, which are initialized in Lines 7 and 8. Lines 11 to 15 represent forward BFS using the iterateInBFS construct, while Lines 16 to 21 represent backward BFS using iterateInReverse (which must be preceded by iterateInBFS).

3.1 Compiler Overview

Figure 1 illustrates how StarPlat's compiler is divided into frontend and backend, just like a general-purpose compiler. Ensuring syntactic and semantic consistency in the StarPlat implementation is the responsibility of the frontend. An abstract syntax tree (AST) of the input code is created by the frontend. Every backend uses the same AST, which is filled during the compiler's parsing phase with the metadata for every construct. After that, it is supplied to the suitable code generator based on the desired target code that the user wants to produce.

```
function ComputeBC (Graph g, propNode <float> BC, SetN<g> sourceSet) {
  g.attachNodeProperty (BC = 0);

  for (src in sourceSet) {
    propNode<float> sigma;
    propNode<float> delta;
    g.attachNodeProperty(delta = 0);
    g.attachNodeProperty(sigma = 0);
    src.sigma = 1;

    iterateInBFS(v in g.nodes() from src){
      for(w in g.neighbors(v)) {
        v.sigma = v.sigma + w.sigma;
      }
    }
    iterateInReverse(v != src) {
      for(w in g.neighbors(v)) {
        v.delta = v.delta + (v.sigma / w.sigma) * (1 + w.delta);
      }
      v.BC = v.BC + v.delta;
} } }
```

Fig 1. BC specification in StarPlat

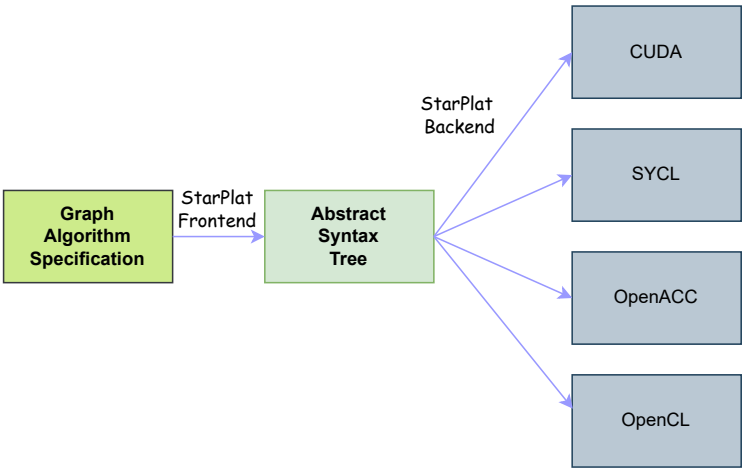

Fig. 1. Process flow of the StarPlat Accelerator compiler

Operator	Reduction Type
+=	Sum
*=	Product
++	Count
&&=	All
||=	Any

Table 1. Reduction operators in StarPlat

3.2 StarPlat Language Constructs

We briefly discuss StarPlat's language constructs [2].

Data Types. StarPlat supports primitive data types: `int`, `bool`, `long`, `float`, and `double`. It also supports Graph, node, edge, node attribute, edge attribute, etc. as first-class types.

Parallelization and Iteration Schemes. `forall` is an aggregate construct that can process a set of elements in parallel. Its sequential counterpart is a simple for statement. Currently, StarPlat supports vertex-based processing.

Reductions. While reduction is often a fundamental building block of parallel languages to enable synchronization, supporting reduction as an extra language construct does not directly align with StarPlat's language design. Therefore, the combined operators such as += are used to specify reduction, preserving the abstraction. The reduction operators supported by StarPlat are tabulated in Table 1.

FixedPoint and Min/Max Constructs. Several solutions to graph algorithms are iterative and converge based on conditions on the node attributes. StarPlat provides a fixedPoint construct to specify this succinctly. Its syntax involves a boolean variable and a boolean expression on node-properties forming the convergence condition, as shown below.

```
fixedPoint until (var: convergence expr) {...}
```

StarPlat provides constructs `Min` and `Max` which perform multiple assignments atomically based on a comparison criterion. This can be useful in update-based algorithms like SSSP, where an update on node properties is carried out on a desired condition while taking care of potential dataraces.

StarPlat also provides aggregate functions `minWt` and `maxWt` to find the minimum and the maximum edge weights.

3.3 Accelerator Programming Backends

We briefly discuss various accelerator backends supported.

CUDA. CUDA is a programming model created by NVIDIA for programming its graphics processing units (GPUs). It is used to speed up computationally demanding activities in a variety of industries, including machine learning, scientific computing, and video game creation. Due to the high computing needs of training deep neural networks, CUDA has grown to be a prominent platform for accelerating deep learning algorithms.

SYCL. SYCL adds data parallelism to C++. It is a standard language for defining data-parallel computations that can be run on many hardware platforms. Several hardware manufacturers, including Intel, AMD, Xilinx, and NVIDIA, support SYCL. SYCL provides a higher-level abstraction that enables programmers to use modern C++ features such as templates and lambda expressions while still writing code in the traditional C++ language. Moreover, SYCL offers a standard library of algebraic and mathematical functions that can be applied to high-performance computing.

OpenCL. Applications that need parallel processing can be accelerated with the help of OpenCL, which offers a consistent programming model. Without needing to write a separate code for every device, it enables developers to create code that can run on a variety of hardware. OpenCL is similar in spirit to StarPlat, but has a limited types of devices supported (e.g., OpenCL cannot parallelize directly in a distributed setting) and cannot take advantage of device-specific features (e.g, warp level intrinsics in NVIDIA GPUs).

OpenACC. Applications can be accelerated with OpenACC on a variety of heterogeneous computing platforms, including GPUs and CPUs. The goal of OpenACC is to offer a high-level parallelization solution that is both portable and performance portable, i.e., the code may run on several hardware platforms and yet deliver satisfactory performance. A programmer can indicate which portions of the code should be parallelized and how they should be parallelized using a set of OpenMP-like directives that are provided by the OpenACC standard. The standard also includes a set of APIs for data management and device management, which allow the programmer to regulate and optimize data transfer between the host and the accelerator (GPU or FPGA).

4 STARPLAT CODE GENERATOR

We augment StarPlat's code generator to support efficient code generation for CUDA, OpenCL, SYCL, and OpenACC. In this section, we discuss key design decisions and the challenges faced during code generation.

4.1 Graph Representation and Storage

Graphs enjoy a variety of representations. In the context of StarPlat, the following requirements drove our decision towards a suitable graph representation.

- should work across all the accelerators, and preferably on CPU too
- should work well with vertex-centric algorithms, common in graph processing
- should be compact (minimizing the extra memory)
- should be fast accessible

The compressed sparse row (CSR) storage format satisfied our requirements well. Since it uses offset-based arrays, the same memory representation works across all the accelerators as well as the CPU. Adjacency matrix was unable to scale to large graphs, adjacency list involved pointer chasing which reduced efficiency, edge-list was compact and fast, but unsuitable for commonly used vertex-based processing.[1]

4.2 Neighborhood Iteration

We discuss accelerator-specific neighborhood traversal below. Figure 2 illustrates it for NVIDIA GPUs. A challenge in CUDA is that the kernel launch and the kernel function are separate. This demands a split-code generation, unlike other backends. This is non-trivial because the StarPlat source code uses simply a `forall` loop. The CUDA code generator needs to identify the variables used in the kernel, transfer those to the device (`cudaMemcpy`) and pass those as parameters to the kernel. The kernel uses the CSR representation (offset array `gpu_OA`) to traverse through the neighbors.

Figure 3 shows neighbor iteration in OpenACC using `#pragma acc` annotations. The code generator does not need to create a separate kernel, but faces a non-triviality similar to that in CUDA, since the variables used within the loop need to be *promoted* up to a `data copyin` clause, to avoid repeated data transfer (e.g., edge weights, vertex attributes).

[1]We advocate edge-list based representation for edge-centric graph processing (e.g., Kruskal's MST) across accelerators.

```
__global__ void computeSSSP (...) {
  unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
  ...
  for (int ee = gpu_OA[id]; ee < gpu_OA[id+1]; ee++) {
    int nbr = g.gpu_edgeList[ee];
    ...
}...}...
computeSSSP<<<nblocks, blocksize >>>(...);
cudaDeviceSynchronize();
```

Fig 2. CUDA code generated for neighborhood iteration

```
#pragma acc data copyin(g)
{
  #pragma acc data copyin( g.edgeList[0:g.num_edges()],
      g.indexofNodes[:g.num_nodes()+1], weight[0: g.num_edges()], modified[0:
      g.num_nodes()], modified_nxt[0:g.num_nodes()]) copy(dist[0:g.num_nodes()])
  { ...
    #pragma acc parallel loop
    for(int v=0; v<g.num_nodes(); v++) {
        for (int ee = g.indexofNodes[v]; ee<g.indexofNodes[v+1]; ee++) {
          int nbr = g.edgeList[ee];
          ...
    }...} ...
} }
```

Fig 3. OpenACC code generated for neighborhood iteration

```
Q.submit([&](handler &h){
  h.parallel_for(NUM_THREADS, [=](id<1> v){
    for (; v < V; v += NUM_THREADS){
      ...
     for(int ee=g.gpu_indexOfNodes[v]; ee<g.gpu_indexOfNodes[v+1];ee++) {
        int nbr = g.gpu_edgeList[ee];
        ...
   }...} ...
}); }).wait()
```

Fig 4. SYCL code generated for neighborhood iteration

Figure 4 shows neighbor iteration in SYCL expressed as a data-parallel kernel using the `parallel_for` function. The first argument, NUM_THREADS, is the number of items to launch in parallel. The second argument is the kernel function to be executed by each work item, which processes up to |V|/NUM_THREADS number of nodes of the graph.

Figure 5 shows neighbor iteration in OpenCL, which, similar to CUDA, has a separate kernel and host processing, with a different syntax. As observed, while the parallelism concepts remain the same, the syntax and the placement of constructs change significantly across the backends.

```
__kernel void computeSSSP(...){
    unsigned id = get_global_id(0);
    ...
    for(int ee = gpu_OA[id]; ee < gpu_OA[id+1]; ee++){
        int nbr = gpu_edgeList[ee];
        ...
    } ...
} ...
// from host
status = clEnqueueNDRangeKernel(command_queue, computeSSSP, 1, NULL, global_work_size,
    local_work_size, 0, NULL, &event);
clWaitForEvents(1, &event);
```

Fig 5. OpenCL code generated for neighborhood iteration

```
int nbr = gpu_edgeList[edge];
int e = edge;
int dist_new = gpu_dist[v] + gpu_weight[e];
if (gpu_dist[nbr] > dist_new) {
    atomicMin(&gpu_dist[nbr], dist_new);
    gpu_modified_next[nbr] = true;
    gpu_finished[0] = false;
}
```

Fig 6. CUDA code generated for reduction / `Min` construct

4.3 Reductions

We discuss accelerator-specific code generation for reduction below. Due to restrictions on the number of resident thread-blocks that might block the kernel, the situation in CUDA becomes challenging. Utilizing a reduction from the host as a distinct kernel (for example, using `thrust::reduce`) is one way to combat this. However, doing so necessitates ending the kernel, leaving the host, calling the reduction kernel, returning to the host, and then starting a new kernel to complete the remaining tasks in the `forall` loop. In addition to complicating code production, this makes processing as a whole, ineffective. Therefore, we rely on CUDA's atomic instructions to generate the functionally equivalent code (e.g., using the function `atomicAdd` as shown in Figure 6 for operator += as used in the triangle counting algorithm).

In the case of OpenACC, the loops with a reduction operation are marked with a `reduction pragma`. The reduction clause in OpenACC is helpful in parallelising in such situations to avoid data races. An example of OpenACC's reduction clause is shown in Figure 7.

Figure 8 shows the usage of reduction in SYCL for Triangle Counting algorithm. It relies on the `atomic_ref` type for achieving data-race-free update to the global count.

Similar to other backends (CUDA and SYCL), OpenCL also uses atomics to implement reduction. Since OpenCL supports atomics only for `int` and `long` types, we have simulated those for `float` and `double` using `atomic_cmpxchg`.

4.4 BFS Traversal

The `iterateInBFS` construct in StarPlat implements Breadth-First Search (BFS) to traverse a graph in parallel. Similar to the case of neighborhood iteration, the CUDA backend expects the code generated for the `iterateInBFS` construct

```
#pragma acc parallel loop reduction(+:diff)
for (int v = 0; v < g.num_nodes(); v++) {
  float sum = 0.0;
  for (int ee = g.rev_indexofNodes[v]; ee < g.rev_indexofNodes[v+1]; ee++) {
    int nbr = g.srcList[ee] ;
    sum = sum + pageRank[nbr] / (g.indexofNodes[nbr+1] - g.indexofNodes[nbr]);
  }
  float val = (1 - delta) / num_nodes + delta * sum;
  diff = diff + val - pageRank[v];
  pageRank_nxt[v] = val;
}
```

Fig 7. Generated OpenACC reduction code in PageRank

```
Q.submit([&](handler &h){
  h.parallel_for(NUM_THREADS, [=](id<1> v){
    for (; v < V; v += NUM_THREADS){
      for (int ee = g.gpu_indexOfNodes[v]; ee < g.gpu_indexOfNodes[v+1]; ee++) {
        int u = g.gpu_edgeList[ee];
        if (u < v){
          ...
          int w = ...
          if (v < w){
            if(findNeighborSorted(u,w,...)){
        atomic_ref<long, memory_order::relaxed, memory_scope::device,
            access::address_space::global_space> atomic_data(dev_triangle_count[0]);
              atomic_data += 1;
  } } } } } }
); }).wait();
```

Fig 8. SYCL code for reduction in Triangle Counting

to be separate for the host and the device. The level-wise BFS kernel on the GPU is called internally by the outer `do-while` loop, which operates on the host (Line 1). The flag (`finished`) is copied across devices (Lines 2 and 6), passed as a parameter to the kernel, and updated in the kernel whenever there is a change in a vertex's level because the loop is on the host.

In both OpenACC and SYCL, the code enters a do-while loop that executes until all the nodes have been visited. In each iteration, the code first sets a boolean variable `finished` to true, indicating that the current level has been processed. It then copies this value to the device memory and launches a kernel function in parallel to process all nodes at the current level. The kernel function iterates through all nodes at the current level and explores their adjacent nodes by iterating over the edges that start at the current node. If an adjacent node has not been visited yet, its level is updated to the current level plus one, and the boolean variable `finished` is set to false, indicating that there are still nodes to be processed. After processing all nodes at the current level, the code updates the level to process in the next iteration and increments the number of hops from the source node. The value of `finished` is then copied back from the device memory to the host memory to check if all the nodes have been visited. The code continues the do-while loop until all the nodes have been visited, which is indicated by the value of the `finished` variable being `true`.

The OpenCL backend code is similar to that in CUDA.

```
do {
  H2D(...);
  BFS<<<...>>>(...)
  cudaDeviceSynchronize();
  ++hops_from_source;
  D2H(...);
  ...
} while (!finished);

__global__ void BFS(...) {
  ...
  if (d_level[u] == *d_hops_from_source) {
    ...
    for (int i=d_offset[u]; i<end; ++i) {
      int v = ...
      if (d_level[v] == -1) {
        d_level[v] = *d_hops_from_source+1;
        *d_finished = false;
} } } }
```

Fig 9. CUDA code for the `iterateInBFS` construct

4.5 Min/Max Constructs

SSSP in StarPlat uses a Min construct as below.

```
<nbr.dist,nbr.modified> = <Min (nbr.dist, v.dist + e.weight), True>;
```

This syntax allows multiple variables (`nbr.modified and nbr.dist`) to be updated atomically.

Here, the for-loop that iterates through all the vertices is parallelized on the accelerator. Each iteration looks for the neighbors of one specific vertex and performs an update operation. When this is being done in parallel, there can be data-races, that is, multiple iterations can write into the distance of the same neighbor at the same time.

The CUDA backend handles Min/Max constructs using the atomic instructions that are readily supported in the language (that is, `atomicMin` and `atomicMax`).

To prevent the data race, OpenACC provides a `#pragma acc atomic write` directive. This directive ensures that no two threads iterations write to this same location at the same time, thereby preventing data races. Figure 10 is the OpenACC generated code for the Min construct in SSSP.[2]

Figure 11 shows the SYCL code generated for `Min`. The conditional update on the node property is achieved through atomic implementations of min and max operations. A relaxed memory ordering is used under which the memory operations can be reordered without any restrictions.

4.6 fixedPoint Construct

The `fixedPoint` construct translates to a while loop conditioned on the fixed-point variable provided in the construct. Typically, the convergence is based on a node property, but it can be an arbitrary computation. This code is generated on the host, so it is similar in template for all the four backends. CUDA code updates a copy of the `finished` flag on the GPU, which is `cudaMemcpy`'ed to the host (as shown in Figure 12).

[2]We observed that the atomic directives do not work reliably on our multicore setup. Hardware and software details are mentioned in Section 6.

```
int dist_new = dist[v] + weight[e];
bool modified_new = true;
if(dist[nbr] > dist_new) {
  int oldValue = dist[nbr];
  #pragma acc atomic write
    dist[nbr] = dist_new;
  modified_nxt[nbr] = modified_new;
  #pragma acc atomic write
      finished = false;
}
```

Fig 10. OpenACC code generated for Min-Construct in SSSP

```
int nbr = gpu_edgelist[edge];
int e = edge;
int dist_new = gpu_dist[v] + gpu_weight[e];
if (gpu_dist[v] != INT_MAX && gpu_dist[nbr] > dist_new) {
  atomic_ref<int, memory_order::relaxed, memory_scope::device,
      access::address_space::global_space> atomic_data(gpu_dist[nbr]);
  atomic_data.fetch_min(dist_new);
  d_modified_next[nbr] = modified_new;
  *d_finished = false;
}
```

Fig 11. SYCL code generated for the `Min` construct

```
while (!finished) {
    finished = true;
    if (...) { // Min-construct expansion
        ...
        finished = false; // fixedPoint identifier updation
}   }
```

Fig 12. Code generated for the `fixedPoint` construct

5 BACKEND OPTIMIZATIONS

We now discuss the accelerator-specific optimizations.

5.1 CUDA and OpenCL

Optimized Host-Device Data Transfer. We run a basic programme analysis on the AST to determine which variables must be transmitted between devices. For instance, since a graph is static, its copy from the GPU to the CPU at the conclusion of the kernel is not necessary. The updated vertex attributes, however, need to be returned. Similar to this, during a fixed-point processing, the `finished` flag is set on the CPU, conditionally set on the GPU, and read on the CPU once again. The variable must therefore be moved back and forth the two devices. Device-only variables are generated for the `forall`-local variables.

Memory Optimization in OR-Reduction. When we write the `fixedPoint` construct, the fixed-point is computed using the `modified` attribute. If any of the vertices' `modified` flags are set, another iteration is fundamentally required.

StarPlat makes use of this, which is essentially a logical-OR operation, to create a single flag variable that is set by multiple threads concurrently (with a reliance on hardware atomicity for primitive types). In terms of time and memory, managing this flag is less expensive than moving arrays of the `modified` flags across devices.

5.2 OpenACC

Optimized Host-Device Data Transfer. In OpenACC, data pragmas have to be generated outside each block after careful analysis of where and when data structures are used in CPU and GPU and when they have to be transferred between CPU and GPU. The data copy between CPU and GPU is a very time-expensive operation and data transfers have to be minimised as much as possible to optimise program run time. Multiple consecutive GPU blocks are combined within a single data transfer so that data are copied less frquently between CPU and GPU. StarPlat performs an analysis to find the necessary graph data variables to be copied to the accelerator before launching the kernel.

Optimized Data Copy around Loops. For GPU parallelised loop, our analysis finds out which variables or arrays need to be copied into GPU at the start of each iteration, which variables need to be copied out into CPU after each iteration and which variables need not be copied in or out. On the basis of this analysis, data `copyin(), copyout(), copy()` pragmas are generated outside the loop.

5.3 SYCL

Optimized Host-Device Data Transfer. The abstract syntax tree (AST) is first analysed in the code-generation process to determine the variables that require a transfer between devices. The graph object is static, thus eliminating the need for constant copying between the GPU and CPU during kernel execution. The graph information is instead transferred to the GPU at the beginning of the function using `malloc_device`. Properties that are modified on the GPU, such as betweenness centrality values or distance from the source for each vertex, must be sent back to the CPU after processing. Additionally, the finished flag is set on the CPU, conditionally set on the GPU, and then read again on the CPU during fixed-point processing. As a result, the variable must be transferred back and forth. Finally, the `forall`-local variables are generated as device-only variables.

Memory Optimization in Reduction. In the StarPlat code-generation process, the `fixedPoint` construct uses a modified property to compute the fixed-point. At a high level, another iteration is required if any of the vertices' modified flags is set, which can be interpreted as a logical-OR operation. To optimize this process, StarPlat generates a single flag variable that is set in parallel by threads. This can be done due to the hardware atomicity for primitive types. The advantage of managing this flag variable is that it is cheaper in terms of time and memory compared to transferring arrays of modified flags across devices.

Computing FixedPoint Efficiently. The fixedPoint construct converges on a specific condition on a single boolean node property. The change of convergence is tracked through a boolean fixed-point variable that ideally needs to be updated after analyzing the property values for all nodes. The update procedure has been optimized by updating the fixed-point variable along with the update to the property value for any node. Since updates to a boolean variable are atomic by hardware, this does not lead to a performance loss.

Graph	Short name	|V| $\times 10^6$	|E| $\times 10^6$	Avg. δ	Max. δ
twitter-2010	TW	21.2	265.0	12.0	302,779
soc-sinaweibo	SW	58.6	261.0	4.0	4,000
orkut	OK	3.0	234.3	76.3	33,313
wikipedia-ru	WK	3.3	93.3	55.4	283,929
livejournal	LJ	4.8	69.0	28.3	22,887
soc-pokec	PK	1.6	30.6	37.5	20,518
usaroad	US	24.0	28.9	2.0	9
germany-osm	GR	11.5	12.4	2.0	13
rmat876	RM	16.7	87.6	5.0	128,332
uniform-random	UR	10.0	80.0	8.0	27

Table 2. Input graphs (δ indicates degree)

	StarPlat	LonestarGPU	Gunrock v 1.2
BC	19	21.2	265.0
PR	17	58.6	261.0
SSSP	15	3.0	234.3
TC	13	3.3	93.3

Table 3. Number of lines for BC, PR, SSSP and TC for Starplat, LonestarGPU and Gunrock v 1.2

6 EXPERIMENTAL EVALUATION

Algorithms. Using StarPlat, we developed four algorithms: Betweenness Centrality (BC), PageRank (PR), Single Source Shortest Path (SSSP), and Triangle Counting (TC), for benchmarking and for comparing with other frameworks and graph libraries. The StarPlat DSL codes for BC and PR fit in 30 lines each while those for SSSP and TC fit in 20. With this small specification, a domain-expert can generate the implementations for these algorithms in different accelerator forms: CUDA, OpenACC, SYCL, OpenCL. Ignoring the header files, the compiler generates around 150, 120, 125, and 75 lines for BC, PR, SSSP, and TC respectively for the CUDA backend. The numbers reduce by about 33% for OpenACC, and increase by 50% and 100% for SYCL and OpenCL. Note that we are using graphs fitting into memory. When that assumption no longer holds, we would need to consider partitioned graphs. Currently, we support vertex-based processing. StarPlat is versatile to support push as well as pull-based models. It is upto the programmer to implement it in a certain way. For instance, SSSP is illustrated with a push-based approach, while Page Rank uses a pull-based (due to dependence on the incoming neighbors).

Graphs. We use ten large graphs in our experiments, which are a mix of different types. Six of these are social networks exhibiting the small-world property, two are road networks having large diameters and small vertex degrees, while two are synthetically generated. One synthetic graph has a uniform random distribution (generated using Green-Marl's graph generator [6]), while the other one has a skewed degree distribution following the recursive-matrix format (generated using SNAP's RMAT generator with parameters a = 0.57, b = 0.19, c = 0.19, d = 0.05) [9]. They are listed in Table 3, sorted on the number of edges in each category. For unweighted graphs, we assign edge-weights selected uniformly at random in the range [1,100] (for SSSP).

Machine Configuration. We ran our experiments for CUDA, OpenCL, and OpenACC on a compute node, whose configuration is as follows: Intel Xeon Gold 6248 CPU with 40 hardware threads spread over two sockets, 2.50 GHz clock, and 192 GB memory running RHEL 7.6 OS. All the codes in C++ are compiled with GCC 9.2, with optimization

Algo.		Framework	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR	Total
BC	1	LonestarGPU	-	-	-	-	-	-	-	-	-	-	-
	1	Gunrock v 1.2	2.122	4.237	0.525	0.535	0.548	0.317	**2.811**	**1.750**	1.238	0.944	15.027
	1	StarPlat	**0.002**	**0.004**	**0.149**	**0.153**	**0.078**	**0.029**	17.656	6.359	**0.225**	**0.079**	24.734
	20	StarPlat	6.992	2.279	2.762	3.014	1.298	0.534	369.701	126.485	2.949	1.593	517.607
	80		28.179	9.332	11.331	12.050	4.886	1.907	1444.656	518.968	6.509	6.372	2044.189
	150		55.548	OOM	21.241	27.271	9.609	3.794	2636.453	978.758	9.912	11.957	—
PR		LonestarGPU	-	**0.240**	0.363	**0.104**	**0.225**	**0.240**	**0.832**	**0.294**	**0.240**	**0.240**	—
		Gunrock v 1.2	15.230	36.910	2.430	2.460	2.952	1.085	13.345	6.499	9.170	5.487	95.568
		StarPlat	**4.081**	7.112	**0.256**	1.780	1.300	0.257	3.420	0.679	0.891	0.257	20.033
SSSP		LonestarGPU	-	0.077	0.217	0.058	0.084	0.037	**0.162**	**0.091**	0.129	0.183	—
		Gunrock v 1.2	2.272	4.057	0.616	0.556	0.562	0.311	1.283	1.140	1.034	0.915	12.746
		StarPlat	**0.001**	**0.002**	**0.078**	**0.044**	**0.027**	**0.012**	1.667	0.695	**0.120**	**0.028**	2.674
TC		LonestarGPU	-	31.990	2.998	2.771	**0.110**	**0.039**	11.874	5.695	**1.270**	0.499	—
		Gunrock v 1.2	**67.718**	7.369	**0.843**	**0.997**	0.850	0.404	1.490	0.712	3.200	1.040	84.623
		StarPlat	10540.002	**1.410**	46.700	4.009	3.006	0.655	**0.001**	**0.001**	824.620	**0.034**	11420.430

Table 4. StarPlat's CUDA code performance comparison against LonestarGPU and Gunrock v 1.2. All times are in seconds. The number in the second column for BC is the number of iterations executed. LonestarGPU does not have BC implemented and fails to load the largest graph TW (OOM == out of memory)

flag -O3. Various backends have the following versions: OpenACC version 20.7-0, OpenCL version 2.0, CUDA version 10.1.243 and run on Nvidia Tesla V100-PCIE GPU with 5120 CUDA cores spread uniformly across 80 SMs clocked at 1.38 GHz with 32 GB global memory and 48 KB shared memory per thread-block. Since SYCL code can run on both CPU and GPU, we ran it on the CPU (with the configuration mentioned above, with 40 threads). However, since the machine does not house Intel GPUs and is not configured to work with its V100 GPUs, we ran it on Intel DevCloud GPU UHD Graphics [0x9a60] and also on another NVIDIA GPU GeForce RTX 2080 Ti.[3]

Baselines. We compare StarPlat-generated accelerator codes against the following two hand-crafted baselines, which also implement the same set of algorithms (except for BC).

- Gunrock [12] provides data-centric abstractions to apply a graph operator on vertices or edges to compute the next frontier using the following three functions to be supplied by the user: filter, compute, and advance. It supports the CUDA backend. By optimizing the processing for improved coalescing and reduced thread-divergence, the Gunrock library constructs efficient implementations of graph processing.
- LonestarGPU [4] is a collection of hand-optimized CUDA programs, with varied optimizations applied for data-driven and topology-driven computations [10].

In the discussion below, we first compare performance of StarPlat against the baselines for the four algorithms on the ten input graphs. Next, we compare the performance of the same algorithm on different accelerator backends.

6.1 Comparison across Frameworks

Our goal is to match the performance of the hand-crafted code. Table 4 presents the absolute running times of the four algorithms on our ten graphs for the three frameworks: LonestarGPU, Gunrock, and StarPlat. The running times include CPU-GPU data transfer (since we will be comparing these times against the CPU-only times as well).

Betweenness Centrality. BC involves running forward and backward shortest paths from various source vertices. Since Brandes algorithm [3] is time-consuming, literature often compares time for a few iterations, spanning a few

[3]Note that SYCL code can run on NVIDIA GPUs by installing a plugin and satisfying certain software and hardware dependencies.

source vertices (otherwise, it can take several days to complete BC on large graphs). The number of iterations is shown against BC in the second column in Table 4. LonestarGPU does not have BC as part of its collection. Compared to Gunrock, StarPlat-generated code outperforms on eight out of the ten graphs. Both the graphs for which Gunrock outperforms are road networks, having large diameter. Gunrock relies heavily on bulk-synchronous processing and its three API are very well optimized. Gunrock's Dijkstra's algorithm works efficiently for road networks. On the other hand, on social and random graphs, our implementation fares better. This is encouraging since StarPlat code is generated. We also illustrate performance with multiple sources of different sizes (20, 80, and 150). Except for soc-sinaweibo, StarPlat-generated code is on par with or better than the other frameworks. Finally, unlike Gunrock and LonstarGPU, StarPlat has the provision to execute BC from a set of source vertices.

PageRank. We observe that the three frameworks have consistent relative performance, with hand-crafted LonestarGPU codes outperforming the other two and StarPlat outperforming Gunrock. StarPlat exploits the double buffering approach to read the current PR values and generate those for the next iteration (see Figure 7). This separation reduces synchronization requirement during the update, but necessitates a barrier across iterations. LonestarGPU uses an in-place update of the PR values and converges faster.

Single-Source Shortest Paths. Gunrock uses Dijkstra's algorithm for computing the shortest paths using a two-level priority queue. We have coded a variant of the Bellman-Ford algorithm in StarPlat. Hence, the comparison may not be most appropriate. But we compare the two only from the application perspective – computing the shortest paths from a source in the least amount of time. LonestarGPU and StarPlat outperform Gunrock on all the ten graphs. Between LonestarGPU and StarPlat, there is no clear winner. They, in fact, have competitive execution times across graphs.

Triangle Counting. Unlike the other three algorithms, TC is not a propagation based algorithm. In addition, it is characterized by a doubly-nested loop inside the kernel (to iterate through neighbors of neighbors). Another iteration is required for checking edge which can be implemented linearly or using binary search if the neighbors are sorted in the CSR representation. Due to this variation in the innermost loop, the time difference across various implementations can be pronounced, which we observe across the three frameworks. Their performances are mixed across the ten graphs, and clearly, StarPlat consumes considerably more time.

6.2 Comparison across Accelerators

Technically, it is inappropriate to compare across different hardware with different software architectures. Nonetheless, we delve into this comparison from end-users' perspective who want their algorithmic code to complete execution as fast as possible. Further, we can potentially compare the performance of various languages (CUDA, OpenACC, OpenCL, SYCL) on the same hardware platform. Table 5 compares performance of the four algorithms across various accelerator codes generated by StarPlat. The backend CUDA is same as StarPlat from Table 4. To the best of our knowledge, this is the first study comparing various accelerators and their languages in the context of graphs algorithms.

Betweenness Centrality. We observe that overall SYCL on NVIDIA GPUs performs the best among the backends with OpenCL on NVIDIA GPUs being a close second. On NVIDIA GPUs, OpenACC performs poorly compared to CUDA, OpenCL, and SYCL. SYCL on NVIDIA GPU outperforms SYCL on Intel GPU. Unlike CUDA, SYCL's implementation does not depend upon `grid.synchronization()`, resulting in better performance on road networks. OpenACC's pragma-based implementation on NVIDIA GPU performs comparable to SYCL on Intel GPU. But on Intel CPU, OpenACC performs poorly compared to SYCL. We also observed that for short diameter graphs, the BC time scales linearly with the number of sources across the backends.

Algo.		Framework	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR	Total
	1	CUDA	**0.01**	**0.01**	**0.15**	0.15	0.08	**0.03**	17.66	6.36	**0.23**	**0.08**	24.73
	1	Openacc(Nvidia GPU)	0.76	1.64	0.57	0.63	0.60	0.12	58.77	21.30	4.29	0.79	89.47
	1	Openacc(Intel CPU)	0.46	1.66	45.15	29.01	76.79	14.06	1670.08	608.26	103.65	59.13	2608.25
	1	OpenCL(Nvidia GPU)	**0.01**	**0.01**	0.17	**0.07**	0.08	**0.03**	3.73	1.37	0.30	0.07	5.83
	1	SYCL(Intel CPU)	0.45	0.51	1.29	1.08	0.78	0.58	71.40	31.09	2.15	1.25	110.58
	1	SYCL(Intel GPU)	0.21	0.29	0.86	0.67	0.35	0.22	57.37	21.22	1.11	1.05	79.35
	1	SYCL(Nvidia GPU)	**0.01**	**0.01**	**0.15**	0.09	**0.07**	**0.03**	**3.02**	**1.31**	0.56	0.10	**5.34**
	20	CUDA	6.99	2.28	2.76	3.01	1.30	0.53	369.70	126.49	2.95	1.59	517.61
	20	Openacc(Nvidia GPU)	16.608	32.6	11.83	12.3	6.42	2.3	1600.2	928.71	86.5	10.81	2708.278
	20	Openacc(Intel CPU)	8.84	84.5	983.61	748.49	1597.88	249.00	30692.97	13267.05	1778.04	1455.57	50865.95
	20	OpenCL(Nvidia GPU)	**0.05**	0.60	**2.12**	**0.96**	1.40	**0.16**	84.42	27.55	**2.28**	**1.33**	120.87
	20	SYCL(Intel CPU)	7.69	1.95	14.76	12.33	7.14	3.1	1611.54	578.56	32.68	16.35	2286.11
	20	SYCL(Intel GPU)	2.40	4.60	13.60	10.98	5.13	2.37	1,122.09	384.42	20.74	18.44	1584.77
	20	SYCL(Nvidia GPU)	0.10	**0.16**	3.06	1.44	**1.11**	0.51	**56.69**	**23.41**	12.22	1.92	**100.60**
BC	80	CUDA	28.179	9.332	11.331	12.050	4.886	1.907	1444.656	518.968	**6.509**	6.372	2044.189
	80	Openacc(Nvidia GPU)	63.78	148.28	**0.35**	51.91	371.58	9.23	7783.67	5644.65	258.06	36.01	14367.52
	80	Openacc(Intel CPU)	51.93	547.75	982.17	3166.13	2730.46	1087.26	96859.19	51268.69	4318.54	10101.74	171113.86
	80	OpenCL(Nvidia GPU)	**0.15**	2.39	8.81	**3.64**	5.22	**0.83**	322.60	111.83	13.54	**5.31**	474.32
	80	SYCL(Intel CPU)	24.18	7.26	58.70	45.62	28.21	11.97	5398.65	2354.16	112.75	59.84	10455.99
	80	SYCL(Intel GPU)	9.12	18.36	50.14	39.63	20.73	8.48	4196.69	1487.65	79.45	62.26	5972.51
	80	SYCL(Nvidia GPU)	0.37	**0.68**	11.43	6.07	**4.38**	1.87	**206.52**	**98.62**	47.89	7.37	**385.2**
		CUDA	4.08	7.11	**0.26**	1.78	1.30	0.26	3.42	0.68	0.89	**0.26**	20.03
		Openacc(Nvidia GPU)	**0.83**	**1.18**	0.63	**0.45**	0.44	0.36	**0.63**	**0.45**	**0.57**	0.70	**6.24**
		Openacc(Intel CPU)	27.60	344.60	4.52	2.60	2.22	0.53	0.95	0.96	7.75	3.11	394.84
PR		OpenCL(Nvidia GPU)	345.59	4.50	33.37	54.13	55.87	19.78	204.97	58.91	439.45	131.88	1357.43
		SYCL(Intel CPU)	70.22	39.91	25.92	13.57	52.95	21.02	347.02	197.28	33.96	47.05	848.90
		SYCL(Intel GPU)	14.57	30.72	4.37	3.28	7.04	2.76	43.38	23.53	7.56	5.69	142.89
		SYCL(Nvidia GPU)	2.70	4.00	1.26	1.19	**0.35**	**0.15**	1.08	0.55	1.20	0.30	12.78
		CUDA	**0.00**	**0.00**	**0.08**	**0.04**	**0.03**	**0.01**	1.67	0.70	**0.12**	**0.03**	**2.67**
		Openacc(Nvidia GPU)	0.72	0.90	0.65	0.42	0.36	0.27	2.8	1.24	0.96	0.48	8.8
SSSP		OpenCL(Nvidia GPU)	**0.00**	**0.00**	**0.08**	0.05	**0.03**	**0.01**	3.72	1.36	**0.12**	**0.03**	5.41
		SYCL(Intel CPU)	2.95	0.73	2.17	1.03	1.28	0.75	53.64	12.15	2.01	1.93	78.63
		SYCL(Intel GPU)	0.82	0.02	0.32	0.12	0.11	0.05	**0.18**	1.24	0.38	0.18	79.39
		SYCL(Nvidia GPU)	4.08	7.11	0.26	1.78	1.30	0.26	3.42	**0.68**	0.89	0.26	6.55
		CUDA	10540.00	1.41	46.70	**4.01**	3.01	0.66	**0.00**	**0.00**	824.62	**0.03**	11420.43
		Openacc(Nvidia GPU)	15522.84	3.23	62.00	6.10	4.98	1.32	0.40	0.35	10770.25	0.45	31625.92
		Openacc(Intel CPU)	20776.85	42.10	3300.97	545.79	225.29	31.58	1.38	0.58	10299.78	15.24	35239.56
TC		OpenCL(Nvidia)	**10162.17**	1.38	48.05	**4.01**	3.08	0.67	**0.00**	**0.00**	791.78	0.04	**11011.17**
		SYCL(Intel CPU)	OOT	21.29	108.86	OOT	11.56	2.21	0.25	0.19	1585.90	0.82	–
		SYCL(Intel GPU)	OOT	11.58	75.12	OOT	8.76	2.89	0.13	0.09	902.38	0.49	–
		SYCL(Nvidia GPU)	OOT	**0.99**	**40.81**	4.04	**2.37**	**0.00**	**0.00**	0.68	**0.89**	0.26	–

Table 5. StarPlat's code performance (seconds) on different accelerators. Second column for BC indicates the number of iterations executed. (OOT == one hour timeout on DevCloud)

PageRank. Unlike in BC, OpenACC on NVIDIA GPUs outperforms the other backends for PR. SYCL on NVIDIA GPUs is a close second, followed by CUDA. OpenCL on NVIDIA GPUs is the slowest (other than on graph GR). On Intel CPU, OpenACC outperforms SYCL on all but one graph (SW). SYCL on Intel GPU outperforms the Intel CPU versions.

Single-Source Shortest Paths. Similar to BC, CUDA outperforms other backends for SSSP. On NVIDIA GPUs, CUDA is followed by OpenCL, then SYCL, and then OpenACC (ignoring two graphs SW and UR for which our generated

OpenCL code did not produce the correct results). We also observed that in our setup OpenACC on multi-core did not produce correct results for SSSP. We nailed it down to the usage of atomics, which we are discussing with NVIDIA.

Triangle Counting. Although only a few lines in the DSL, TC is a time-consuming algorithm. This is evident from SYCL going out of time (one hour timeout on DevCloud) on two graphs (TW and OK). Otherwise, SYCL performs well on almost all the graphs. On NVIDIA GPUs, CUDA and OpenCL perform similarly, while OpenACC is twice as slow. On Intel CPU, SYCL considerably outperforms OpenACC. SYCL on Intel GPU is comparable to that on Intel CPU. Among all the graphs, TW and RM stand-out for extremely high running times. This is due to (i) a large number of triangles in them, and (ii) their skewed degree distribution.

7 RELATED WORK

Graph algorithms have been mostly explored in CUDA.

Gunrock [12] is a graph library which uses data-centric abstractions to perform operations on edge and vertex frontiers. All the Gunrock operations are bulk-synchronous, and they affect the frontier by operating on the values within it or by computing a new one, using the following three functions: *filter*, *compute*, and *advance*. Gunrock library constructs efficient implementations of frontier operations with coalesced accesses and minimal thread divergence in CUDA. LonestarGPU [4] is a collection of graph analytic CUDA programs. It employs multiple techniques related to computation, memory, and synchronization to improve performance of the underlying graph algorithms. We quantitatively compare StarPlat against Gunrock and LonestarGPU.

Medusa [14] is a software framework which eases the work of GPU computation tasks. Similar to Gunrock, it provides APIs to build upon, to construct various graph algorithms. Medusa exploits the BSP model, and proposes a new model EVM (Edge Message Vertex), wherein the local computations are performed on the vertices and the computation progresses by passing messages across edges. CuSha [8] is a graph processing framework that uses two graph representations: G-Shards and Concatenated Windows (CW). G-Shards makes use of a recently developed idea for non-GPU systems that divides a graph into ordered sets of edges called as *shards*. In order to increase GPU utilisation for processing sparse graphs, CW is a new format that improves the use of shards. CuSha improves GPU utilization by processing several shards in parallel on the streaming multiprocessors. CuSha's architecture for parallel processing of large graphs allows the user to create the vertex-centric computation and plug it in, making programming easier. CuSha is demonstrated to significantly outperform the virtual warp-centric approach. MapGraph [5] is a parallel graph programming framework which provides a high level abstraction, which helps in writing efficient graph programs. It uses SOA (Structure Of Arrays) to ensure coalesced memory accesses. It uses the dynamic scheduling strategy using GAS (Gather-Apply-Scatter) abstraction. Dynamic scheduling improves the memory performance and dispense the workload to the threads in accordance with degree of vertices.

T. Hoshino et al. [7] presents an early comparison of OpenACC and CUDA performance using two small bencharks: stencil and matrix multiplication. They also present a real-world CFD application benchmark for comparison. Sandra Wienke et al. [13] showed a study of OpenACC and compared its performance against OpenCL on two real world applications: Simulation of bevel gear cutting and Neuromagnetic Inverse Problem. They conclude that OpenACC offers a promising development effort to performance ratio based on these benchmarks. K. Alsubhi et al. [1] proposed a tool to translate sequential C++ code into OpenACC parallelised code. The tool used an analyser that detects blocks that could be run in parallel, finds data dependency between different blocks and finds the type of parallelism that could be used.

Tomusk [11] makes a case for using OpenCL for performing computations on sparse graph algorithms, and claims that OpenCL is expressive enough to support custom optimizations for various graph-analytic algorithms.

8 CONCLUSION

We illustrated that it is feasible to generate efficient parallel code for multiple accelerators from the same algorithmic specification in a graph DSL. This is not only viable, but is also a desirable approach especially for domain-experts who are currently forced to learn multiple languages for accelerating their scientific computation. A limitation of StarPlat is that it is still a new language. While writing code in the StarPlat DSL is relatively easy and short, it would be helpful if the learning curve can be further reduced.

ACKNOWLEDGMENTS

We gratefully acknowledge the use of the computing resources at HPCE, IIT Madras. This work is supported by grants from KLA and India's National Supercomputing Mission.

REFERENCES

[1] K. Alsubhi, F. Alsolami, A. Algarni, E. Albassam, M. Khemakhem, F. Eassa, K. Jambi, and M. Usman Ashraf. A tool for translating sequential source code to parallel code written in c++ and openacc. In *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, New York, 2019. IEEE.

[2] Nibedita Behera, Ashwina Kumar, Ebenezer Rajadurai T, Sai Nitish, Rajesh Pandian M, and Rupesh Nasre. StarPlat: A Versatile DSL for Graph Analytics, 2023.

[3] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. In *Journal of Mathematical Sociology*, volume 25, pages 163–177, 2001.

[4] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization, IISWC 2012, La Jolla, CA, USA, November 4-6, 2012*, pages 141–151, New York, NY, USA, 2012. IEEE Computer Society.

[5] 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, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *SIGPLAN Not.*, 47(4):349–362, mar 2012.

[7] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143, New York, 2013. IEEE.

[8] 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, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Jure Leskovec and Rok Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[10] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 463–474, New York, 2013. IEEE.

[11] Erik Tomusk. Executing Graphs with OpenCL. In *International Workshop on OpenCL*, IWOCL'21, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: a high-performance graph processing library on the GPU. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 11:1–11:12, Barcelona, Spain, 2016. ACM.

[13] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc — first experiences with real-world applications. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 859–870, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.