A Commodity Performance Baseline for HIVE Graph Applications: Year 1 Report

Ben Johnson Weitang Liu Agnieszka Łupińska
Muhammad Osama John D. Owens Yuechao Pan
Leyuan Wang Xiaoyun Wang Carl Yang

UC Davis

Contents

Chapter 1

Executive Summary

This report is located online at the following URL: https://gunrock.github.io/docs/hive_year1_summary.html.

Herein UC Davis produces the following three deliverables that it promised to deliver in Year 1:

1. **7–9 kernels running on a single GPU on DGX-1**. The PM had indicated that the application targets are the graph-specific kernels of larger applications, and that our effort should target these kernels. These kernels run on one GPU of the DGX-1. These kernels are in Gunrock's GitHub repository as standalone kernels. While we committed to delivering 7–9 kernels, we deliver all 11 v0 kernels.
2. **(High-level) performance analysis of these kernels**. In this report we analyze the performance of these kernels.
3. **Separable communication benchmark predicting latency and throughput for a multi-GPU implementation**. This report (and associated code, also in the Gunrock GitHub repository) analyzes the DGX-1's communication capabilities and projects how single-GPU benchmarks will scale on this machine to 8 GPUs.

Specific notes on applications and scaling follow:

Application Classification Application classification involves a number of dense-matrix operations, which did not make it an obvious candidate for implementation in Gunrock. However, our GPU implementation using the CUDA CUB library shows substantial speedups (10-50x) over the multi-threaded OpenMP implementations.

However, there are two neighbor reduce operations that may benefit from the kind of load balancing implemented in Gunrock. Thus, it would be useful to either expose lightweight wrappers of high-performance Gunrock primitives for easy intergration into outside projects *or* come up with a workflow inside of

Gunrock that makes programming applications with lots of non-graph operations straightforward.

Geolocation Geolocation or geotagging is an interesting parallel problem, because it is among the few that exhibits the dynamic parallism pattern within the compute. The pattern is as follows; there is parallel compute across nodes, each node has some serial work and within the serial work there are sveral parallel math operations. Even without leveraging dynamic parallelism within CUDA (kernel launches within a kernel), Geolocation performs well on the GPU environment because it mainly requires simple math operations, instead of complicated memory movement schemes.

However, the challenge within the application is load balancing this simple compute, such that each processor has roughly the same amount of work. Currently, in gunrock, we map Geolocation using the `ForAll()` compute operator with optimizations to exit early (performing less work and fewer reads). Even without addressing load balancing issue with a complicated balancing scheme, on the HIVE datasets we achieve a 100x speedup with respect to the CPU reference code, implemented using C++ and OpenMP, and ~533x speedup with respect to the GTUSC implementation. We improve upon the algorithm by avoiding a global gather and a global synchronize, and using 3x less memory than the GTUSC reference implementation.

`GraphSearch` Graph search is relatively minor modification to Gunrock's random walk application, and was straightforward to implement. Though random walks are a "worst case scenario" for GPU memory bandwidth, we still achieve 3-5x speedup over a modified version of the OpenMP reference implementation.

The original OpenMP reference implementation actually ran slower with more threads – we fixed the bugs, but the benchmarking experience highlights the need for performant and hardened CPU baselines.

Until recently, Gunrock did not support parallelism *within* the lambda functions run by the `advance` operator, so neighbor selection for a given step in the walk is done sequentially. Methods for exposing more parallism to the programmer are currently being developed via parallel neighbor reduce functions.

In an end-to-end graph search application, we'd need to implement the scoring function as well as the graph walk component. For performance, we'd likely want to implement the scoring function on the GPU as well, which makes this a good example of a "Gunrock+X" app, where we'd need to integrate the high-performance graph processing component with arbitrary user code.

Local Graph Clustering (LGC) This variant of local graph clustering (L1 regularized PageRank via FISTA) is a natural fit for Gunrock's frontier-based programming paradigm. We observe speedups of 2-3 orders of magnitude over the HIVE reference implementation.

The reference implementation of the algorithm was not explicitly written as `advance`/`filter`/`compute` operations, but we were able to quickly determine

how to map the operations by using a lightweight Python implementation of the Gunrock programming API as a development environment. Thus, LGC was a good exercise in implementing a non-trivial end-to-end application in Gunrock from scratch.

Graph Projections Because it has a natural representation in terms of sparse matrix operations, graph projections gave us an opportunity to compare ease of implementation and performance between Gunrock and another UC-Davis project, GPU GraphBLAS.

Overall, we found that Gunrock was more flexible and more performant than GraphBLAS, likely due to better load balancing. However, in this case, the GraphBLAS application was substantially easier to program than Gunrock, and also allowed us to take advantage of some more sophisticated memory allocation methods available in the GraphBLAS cuSPARSE backend. These findings suggest that addition of certain commonly used API functions to Gunrock could be a fruitful direction for further work.

Seeded Graph Matching (SGM) SGM is a fruitful workflow to optimize, because the existing implementations were not written with performance in mind. By making minor modifications to the algorithm that allow use of sparse data structures, we enable scaling to larger datasets than previously possible.

The application involves solving a linear assignment problem (LSAP) as a subproblem. Solving these problems on the GPU is an active area of research – though papers have been written describing high-performance parallel LSAP solvers, reference implementations are not available. We implement a GPU LSAP solver via Bertsekas' auction algorithm, and make it available as as standalone library.

SGM is an approximate algorithm that minimizes graph adjacency disagreements via the Frank-Wolfe algorithm. Certain uses of the auction algorithm can introduce additional approximation in the gradients of the Frank-Wolfe iterations. An interesting direction for future work would be a rigorous study of the effects of this kind of approximation on a variety of different graph tolopogies. Understanding of those dynamics could allow further scaling beyond what our current implementations can handle.

Vertex Nomination The term "vertex nomination" covers a variety of different node ranking schemes that fuse "content" and "context" information. The HIVE reference code implements a "multiple-source shortest path" context scoring function, but uses a very suboptimal algorithm. By using a more efficient algorithm, our serial CPU implementation achieves 1-2 orders of magnitude speedup over the HIVE implementation and our GPU implementation achieves another 1-2 orders of magnitude on top of that. Implementation was straightforward, involving only a small modification to the existing Gunrock SSSP app.

Chapter 2

Application Classification

The application classification (AC) workflow is an implementation of probabalistic graph matching via belief propagation. The workflow takes two node- and edge-attributed graphs as input – a data graph `G = (U_G, E_G)` and a pattern graph `P = (U_P, E_P)`. The goal is to find a subgraph `S` of `G` such that the dissimilarity between the node/edge features of `P` and `S` is minimized. The matching is optimized via loopy belief propagation, which consists of iteratively passing messages between nodes then updating beliefs about the optimal match.

Summary of Results

Application classification involves a number of dense-matrix operations, which did not make it an obvious candidate for implementation in Gunrock. However, our GPU implementation using the CUDA CUB library shows substantial speedups (10-50x) over the multi-threaded OpenMP implementations.

However, there are two neighbor reduce operations that may benefit from the kind of load balancing implemented in Gunrock. Thus, it would be useful to either expose lightweight wrappers of high-performance Gunrock primitives for easy intergration into outside projects *or* come up with a workflow inside of Gunrock that makes programming applications with lots of non-graph operations straightforward.

Summary of CUDA Implementation

We implement application classification from scratch in CUDA using CUB rather than Gunrock. Application classification requires the following kernels:

- compute pairwise distance between rows of dense matrices
- normalize the columns of a dense matrix
- compute maximum of columns in a dense matrix
- gather/scatter rows of dense matrix

- sum/max neighborhood reduce on the data/pattern graphs

Apart from the last one, these kernels do not obviously map to Gunrock's advance/filter model.

Pseudocode for the core belief propagation algorithm is as follows:

```
for iteration in range(num_iterations):

    # Update edge messages
    edge_msg_f = diff_r[data_edges.srcs] - edge_distances # data.num_edges x pattern.num_ed
    edge_msg_r = diff_f[data_edges.srcs] - edge_distances # data.num_edges x pattern.num_ed

    # Normalize edge messages
    edge_msg_f = normprob(edge_msg_f)
    edge_msg_r = normprob(edge_msg_r)

    # Max of forward/backward messages
    f_null = columnwise_max(diff_f)                          # 1 x pattern.num_edges
    r_null = columnwise_max(diff_r)                          # 1 x pattern.num_edges
    for edge_idx, (src, dst) in enumerate(data_edges):
        max_r[src] = max(max_r[src], edge_msg_r[edge_idx], f_null)
        max_f[dst] = max(max_f[dst], edge_msg_f[edge_idx], r_null)

    # Update beliefs
    belief = - node_distances                                # data.num_nodes x patt.num_node
    for edge_idx, (src, dst) in enumerate(pattern_edges):
        belief[:,dst] += max_f[:,edge_idx]
        belief[:,src] += max_r[:,edge_idx]

    # Normalize beliefs
    belief = normprob(belief)

    diff_f = belief[:,pattern_edges.dsts] - max_f          # data.num_nodes x pattern.num_e
    diff_r = belief[:,pattern_edges.srcs] - max_r          # data.num_nodes x pattern.num_e
```

where `normprob` is the column-wise log-softmax function. That is, since `belief` is `data.num_nodes x patt.num_nodes`, each node in the `pattern` graph gets assigned a probability distribution over nodes in the `data` graph.

Our implementation is based on the PNNL implementation rather than the distributed GraphX reference implementation. On all of the graphs we've tested, the output of our implementation exactly matches the output of the PNNL code. According to PNNL, their implementation may give different results than the HIVE reference implementation (due to e.g., different normalization schemes).

How To Run This Application on DARPA's DGX-1

Prereqs/input

```
git clone --recursive \
        https://github.com/owensgroup/application_classification
cd application_classification
make clean
make
```

Running the application

Example Command

```
mkdir -p results
./main \
    ./data/georgiyData.Vertex.csv \
    ./data/georgiyData.Edges.csv \
    ./data/georgiyPattern.Vertex.csv \
    ./data/georgiyPattern.Edges.csv > results/georgiy
```

Example output

```
$ head -n 5 results/georgiy
-8.985810e+00
-3.859019e+01
-4.470994e+01
-1.673157e+01
-1.730952e+01
$ tail -n 5 results/georgiy
-2.886186e+01
-1.499165e+01
-4.034595e+01
-1.060496e+01
-7.684015e+00
$ cat results/georgiy | openssl md5
(stdin)= bd57a5126d5f943ad5c15408d410790d
```

Expected Output

The output of the program is a `data.num_nodes x pattern.num_nodes` matrix, where each column represents a log-probability distribution of assignments of pattern node `j` to data node `i`. This matrix is printed in row major order as a file with `data.num_nodes x pattern.num_nodes` lines.

In python, you could inspect the output like:

```
import numpy as np

# load results
x = open('./results/georgiy').read().splitlines()
x = [float(xx) for xx in x]

# reshape to (data.num_nodes, pattern.num_nodes)
x = np.reshape(x, (1000, 10))

# exponentiate
x = np.exp(x)

# columns should sum to 1
x.sum(axis=0)
# array([1.0000001 , 1.00000001, 1.00000004, 0.99999999, 0.99999988,
       # 0.99999994, 0.99999985, 1.00000001, 1.        , 0.99999988])
```

As previously mentioned, our output exactly matches the output of the PNNL implementation on all of the graphs we have tested.

Performance and Analysis

We measure performance by runtime of the algorithm given

- a node- and edge-attributed data graph
- a node- and edge-attributed pattern graph

Though AC computes probabilities of matches between data and pattern nodes, this is a deterministic and parameterless algorithm, so we do not measure performance in terms of accuracy. Further, runtime does not depend on the *values* of the node/edge attributes, so we can reasonably run experiments using randomly generated values.

Implementation limitations

As currently implemented, the algorithm allocates a number of arrays of floats:

- 3 arrays of size `data.num_edges * pattern.num_edges`
- 3 arrays of size `data.num_nodes * pattern.num_edges`
- 2 arrays of size `data.num_nodes * pattern.num_nodes`

The combined memory usage of these arrays will be the primary bottleneck for scaling. For example, if

- data.num_nodes = 6M
- data.num_edges = 18M
- pattern.num_nodes = 10
- pattern.num_edges = 20

then the memory footprint will be on the order of 13 GB. (*Note:* It's likely that reordering of operations could reduce the number of intermediate data structures required.)

AC is only applicable to graphs with node and edge features. However, the runtime of the algorithm is only dependent on the *dimension* of these features, rather than the values. Thus, for benchmarking purposes, we can pick a `node_feature_dim` and `edge_feature_dim` and use randomly generated features. This is helpful becaus we do not have a good real-world dataset for benchmarking.

The algorithm requires both a data graph and a smaller pattern graph. Both the size and topology of the pattern graph may affect the runtime of the algorithm. However, we do not have examples of actual pattern graphs apart from `georgiyPattern`, so we are forced to generate them synthetically. Specifically, we do this by sampling a node `q`, sampling up to `max_pattern_nodes` number of `u` to form set `Q`, and using the subgraph induced by `Q + q` as the pattern graph.

We suspect the PNNL implementation may have a couple of minor bugs:

- the `CV` matrix is log-softmax normalized in the initialization phase, updated with values from `FMax` and `RMax`, and log-softmax normalized again. This kind of double normalization seems strange, and is perhaps incorrect.
- In our `RepeatColumnsByDataEdges`, `FE` and `RE` both use the `src` of an edge, which conflicts with the algorithm description in the paper. One of these should probably be using the `src` and the other the `dst`.

These bugs are easy to fix, but we left them "as is" because a) we have no easy way to verify which is correct and b) we'd like consistency of results w/ the PNNL implementation.

Comparison against existing implementations

The original reference implementation consisted of a large amount of distributed Spark and GraphX Scala code. For ease of implementation, and to make sure performance comparisons are meaningful, we instead based our implementation on the PNNL ApplicationClassification OpenMP code.

Overall, the CUDA implementation is between 10x and 100x faster than the best run of the PNNL OpenMP code.

We compare our CUDA implementation to PNNL's C++ OpenMP implementation on several different graphs:

georgiyData

- a small graph included w/ real (source unknown) node/edge features (included in PNNL repo)
- `|U|=1000 |E|=20135 node_feat_dim=13 edge_feat_dim=17`

rmat18

- a scale 18 RMAT synthetic graph w/ random node/edge features (included in PNNL repo)
- `|U|=262145 |E|=2008660 node_feat_dim=13 edge_feat_dim=17`

JohnsHopkins_random

- Social network graph w/ random node/edge features
- `|U|=5157 |E|=186572 node_feat_dim=12 edge_feat_dim=16`

For the first two, we use the `georgiyPattern` pattern graph included in the PNNL repo. For the latter, we generate a pattern graph using the neighborhood induction mentioned above.

georgiyData

implementation	threads	elapsed_ms
PNNL OpenMP	1	1635.774136
PNNL OpenMP	2	1405.072927
PNNL OpenMP	4	1005.914927
PNNL OpenMP	8	831.342936
PNNL OpenMP	16	793.069839
PNNL OpenMP	32	546.305180
PNNL OpenMP	64	706.761122
Our CUDA	1xP100	**42.533**

Takeaway: Our CUDA implementation is approximately 12.8x faster than the fastest OpenMP run (32 threads). However, this problem is quite small and both implementations run in under 1 second.

rmat18

implementation	threads	elapsed_ms
PNNL OpenMP	1	113337.949038
PNNL OpenMP	2	142036.607981

implementation	threads	elapsed_ms
PNNL OpenMP	4	109564.634800
PNNL OpenMP	8	95680.689096
PNNL OpenMP	16	87083.579063
PNNL OpenMP	32	88772.798061
PNNL OpenMP	64	82495.028973
Our CUDA	1xP100	**827.573**

Takeaway: Our CUDA implementation is approximately 99x faster than the fastest PNNL run (64 threads). The absolute magnitude of the differences is much more substantial here – the CUDA implementation runs in < 1 second while the OpenMP version runs in ~ 1.5 minutes.

JohnsHopkins_random

implementation	threads	elapsed_ms
PNNL OpenMP	1	71190.566063
PNNL OpenMP	2	44293.390989
PNNL OpenMP	4	31736.392021
PNNL OpenMP	8	24292.662144
PNNL OpenMP	16	21556.239128
PNNL OpenMP	32	17082.650900
PNNL OpenMP	64	19473.608017
Our CUDA	1xP100	**450.897**

Takeaway: Our CUDA implementation is approximately 37x faster than the fastest PNNL run (32 threads). Again, the absolute difference in runtimes is more substantial, with our code running in < 1 second vs. ~ 20 seconds.

Performance limitations

Profiling (on the RMAT graph) reveals that the distribution of runtime over kernels is flatter in AC than for many applications – often, a single kernel will account for $> 80\%$ of runtime, but here the most expensive kernel only accounts for 12.8% of compute time.

- 12.8% of time spent in `__reorderColumns`, which is a gather/scatter operation (memory bandwidth = 281 GB/s)
- 10.5% of time spent in `__transpose`
- 12.1% of time spend in `__rowSubLog`, an edgewise arithmetic operation
- 9.5% of time computing `data.num_edges x pattern.num_edges` similarity matrix

- 8.1% of time doing neighborhood reductions (293 GB/s)

All of these are memory bound operations.

As mentioned above, the memory use of the app could be reduced (if need be) by reducing the number of intermediate data structures. This would come at the cost of increased (re)computation time.

Next Steps

Alternate approaches + Gunrock implications

In the CUDA implementation, there are a number of places where we could take advantage of multiple streams to reduce runtimes. For example, in the initialization phase, we compute the pairwise distance between a) data/pattern node features and b) data/pattern edge features. These operations are completely independent of one another, and so could happen asynchronously on different streams.

The application was implemented outside of the Gunrock framework because it had a large number of (dense matrix) operations that are not explicitly supported by Gunrock, and a relatively small number of kernels that map well to Gunrock's advance/filter paradigm. However, the code uses CUB's `DeviceSegmentedReduce` a number of times – Gunrock recently added a similar operator that is load-balanced for better performance. In the future, it would be worthwhile to see what kind of speedup we could get from the Gunrock version, which should roughly be a drop-in replacement.

Notes on multi-GPU parallelization

Most of the kernels are either row or column operations (reductions) over dense matrices, and thus relatively easy to partition over multiple nodes. They would either end up being embarrassingly parallel or would require a per-node reduction and then a reduction across nodes. Replacing CUB's `DeviceSegmentedReduce` with Gunrock's implementation would give us multi-GPU support for the remaining kernel.

Alternatively, depending on the topology of the graph, we may be able to partition the data graph so that we can duplicate the pattern graph across nodes and run an independent instance of application classification on each partition. The partition would need to be constructed in a way that ensures that every subgraph is intact on *some* GPU, which implies some partial duplication of the data graph. If the data graph has a large diameter, or the pattern graph has a small diameter, this may be possible without excessive duplication. If the data graph has a small diameter, we may still be able to partition the graph by e.g. removing edges that are particularly dissimilar from edges in the pattern graph. This kind of approach is clearly very application specific, and may not be possible at all in some cases.

Notes on dynamic graphs

In practice, it's likely that practitioners would like to run application classification on a dynamic graph (e.g., the HIVE use case was for detecting certain suspicious patterns of communication in a cybersecurity graph). However, it is not obvious how the current algorithm would be applied in a streaming fashion without relatively major modifications. It's more likely that the current AC algorithm would be applied to a dynamic graph via some kind of sliding window.

Notes on larger datasets

We may be able to use a partitioning scheme like the one described in the multi-GPU section above to handle data graphs that are larger than GPU memory.

Notes on other pieces of this workload

The documentation on the wiki includes discussion of the various featurization methods used to produce the node/edge attributes. These are beyond the scope of this work, but do include things such as computing degree, number of triangles, betweenness centrality, etc. If we wanted to build a high-performance end-to-end application classification system, we would want to implement some of these featurization methods in Gunrock as well.

Chapter 3

Geolocation

Infers user locations using the location (latitude, longitude) of friends through spatial label propagation. Given a graph `G`, geolocation examines each vertex `v`'s neighbors and computes the spatial median of the neighbors' location list. The output is a list of predicted locations for all vertices with unknown locations.

Summary of Results

Geolocation or geotagging is an interesting parallel problem, because it is among the few that exhibits the dynamic parallism pattern within the compute. The pattern is as follows; there is parallel compute across nodes, each node has some serial work and within the serial work there are sveral parallel math operations. Even without leveraging dynamic parallelism within CUDA (kernel launches within a kernel), Geolocation performs well on the GPU environment because it mainly requires simple math operations, instead of complicated memory movement schemes.

However, the challenge within the application is load balancing this simple compute, such that each processor has roughly the same amount of work. Currently, in gunrock, we map Geolocation using the `ForAll()` compute operator with optimizations to exit early (performing less work and fewer reads). Even without addressing load balancing issue with a complicated balancing scheme, on the HIVE datasets we achieve a 100x speedup with respect to the CPU reference code, implemented using C++ and OpenMP, and ~533x speedup with respect to the GTUSC implementation. We improve upon the algorithm by avoiding a global gather and a global synchronize, and using 3x less memory than the GTUSC reference implementation.

Summary of Gunrock Implementation

There are two approaches we took to implement Geolocation within gunrock:

- **[Fewer Reads] Global Gather:** uses two `compute` operators as `ForAll()`. The first `ForAll()` is a `gather` operation, gathering all the values of neighbors with known locations for an active vertex `v`, and the second `ForAll()` uses those values to compute the `spatial_center` where the spatial center of a list's points is the center of those points on the earth's surface.

```
def gather_op(Vertex v):
    for neighbor in G.neighbors(v):
        if isValid(neighbor.location):
            locations_list[v].append(neighbor.location)

def compute_op(Vertex v):
    if !isValid(v.location):
        v.location = spatial_center(locations_list[v])
```

- **[Less Memory] Repeated Compute:** skips the global gather and uses only one `compute` operator as a `ForAll()` to find the spatial center of every vertex. During the spatial center computation, instead of iterating over all valid neighbors (where valid neighbor is a neighbor with a known location), we iterate over all neighbors for each vertex, doing more random reads than the global gather approach, but using `3x` less memory.

```
def spatial_center(Vertex v):
    if !isValid(v.location):
        v.location = spatial_median(neighbors_list[v])
```

- **[Optimization] Early Exit:** fuses the global gather approach with the repeated compute, by performing one local gather for every vertex within the spatial center operator (without a costly device barrier), and exiting early if a vertex `v` has only one or two valid neighbors:

```
def spatial_center(Vertex v):
    if !isValid(v.location):
        if v.valid_locations == 1:
            v.location = valid_neighbor[v].location:
            exit
        else if v.valid_locations == 2:
            v.location = mid_point(valid_neighbors[v].location)
        else:
            v.location = spatial_median(neighbors_list[v])
```

Approach	Memory Usage	Memory Reads/Vertex	Device Barriers	Largest Dataset (P100)
Global Gather	O(3x|E|)	# of valid locations	1	~160M Edges
Repeated Compute	O(|E|)	degree of vertex	0	~500M Edges

Note: `spatial_median()` is defined as center of points on earth's surface – given a set of points `Q`, the function computes the point `p` such that: `sum([haversine_distance(p, q) for q in Q])` is minimized. See `gunrock/app/geo/geo_spatial.cuh` for details on the spatial median implementation.

How To Run This Application on DARPA's DGX-1

Prerequisites

```
git clone --recursive https://github.com/gunrock/gunrock -b dev-refactor
cd gunrock
mkdir build
ctest ..
cd ../tests/geo/
make clean && make
```

HIVE Data Preparation

Prepare the data, skip this step if you are just running the sample dataset. Assuming we are in `tests/geo` directory:

```
export TOKEN= # get this Authentication TOKEN from
              # https://api-token.hiveprogram.com/#!/user
wget --header "Authorization:$TOKEN" \
  https://hiveprogram.com/data/_v0/geotagging/instagram/instagram.tar.gz
tar -xzvf instagram.tar.gz && rm instagram.tar.gz
cd instagram/graph
cp ../../generate-data.py ./
python generate-data.py
```

This will generate two files, `instagram.mtx` and `instagram.labels`, which can be used as an input to the geolocation app.

Running the application

Application specific parameters:

```
--labels-file
    file name containing node ids and their locations.

--geo-iter
    number of iterations to run geolocation or (stop condition).
    (default = 3)

--spatial-iter
    number of iterations for spatial median computation.
    (default = 1000)

--geo-complete
    runs geolocation for as many iterations as required
    to find locations for all nodes.
    (default = false because it uses atomics)

--debug
    Debug label values, this prints out the entire labels
    array (longitude, latitude).
    (default = false)
```

Example command-line:

```
# geolocation.mtx is a graph based on chesapeake.mtx dataset
./bin/test_geo_10.0_x86_64 --graph-type=market --graph-file=./geolocation.mtx \
  --labels-file=./locations.labels --geo-iter=2 --geo-complete=false
```

Sample input (labels):

```
% Nodes Latitude Longitude
39 2 2
1 37.7449063493 -122.009432884
2 37.8668048274 -122.257973253
4 37.869112506 -122.25910604
6 37.6431858915 -121.816156983
11 37.8652346572 -122.250634008
19 38.2043433677 -114.300341275
21 36.7582225593 -118.167916598
22 33.9774659389 -114.886512278
30 39.2598884729 -106.804662071
31 37.880443573 -122.230147039
39 9.4276164485 -110.640705659
```

Sample output:

```
Loading Matrix-market coordinate-formatted graph ...
Reading from ./geolocation.mtx:
  Parsing MARKET COO format edge-value-seed = 1539674096
 (39 nodes, 340 directed edges)...
Done parsing (0 s).
  Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...
Done converting (0s).
Labels File Input: ./locations.labels
Loading Labels into an array ...
Reading from ./locations.labels:
  Parsing LABELS
 (39 nodes)
Done parsing (0 s).
Debugging Labels -------------
 (nans represent unknown locations)
    locations[ 0 ] = < 37.744907 , -122.009430 >
    locations[ 1 ] = < 37.866806 , -122.257973 >
    locations[ 2 ] = < nan , nan >
    locations[ 3 ] = < 37.869114 , -122.259109 >
     ...
    locations[ 35 ] = < nan , nan >
    locations[ 36 ] = < nan , nan >
    locations[ 37 ] = < nan , nan >
    locations[ 38 ] = < 9.427616 , -110.640709 >

---------------------------
______ CPU Reference _____
---------------------------
 Elapsed: 0.267029
Initializing problem ...
Number of nodes for allocation: 39
Initializing enactor ...
Using advance mode LB
Using filter mode CULL
nodes=39

---------------------------
0        0       0       queue3          oversize :      234 ->  342
0        0       0       queue3          oversize :      234 ->  342
---------------------------
Run 0 elapsed: 11.322021, #iterations = 2
Node [ 0 ]: Predicted = < 37.744907 , -122.009430 > Reference = < 37.744907 , -122.009430 >
Node [ 1 ]: Predicted = < 37.866806 , -122.257973 > Reference = < 37.866806 , -122.257973 >
Node [ 2 ]: Predicted = < 9.427616 , -110.640709 > Reference = < 9.427616 , -110.640709 >
Node [ 3 ]: Predicted = < 37.869114 , -122.259109 > Reference = < 37.869114 , -122.259109 >
...
Node [ 35 ]: Predicted = < 37.864429 , -122.199409 > Reference = < 37.864429 , -122.199409
Node [ 36 ]: Predicted = < 23.755602 , -115.803055 > Reference = < 37.807079 , -122.134163
```

```
Node [ 37 ]: Predicted = < 37.053715 , -115.913658 > Reference = < 37.053719 , -115.913628
Node [ 38 ]: Predicted = < 9.427616 , -110.640709 > Reference = < 9.427616 , -110.640709 >
0 errors occurred.
[geolocation] finished.
 avg. elapsed: 11.322021 ms
 iterations: 2
 min. elapsed: 11.322021 ms
 max. elapsed: 11.322021 ms
 load time: 68.671 ms
 preprocess time: 496.136000 ms
 postprocess time: 0.463009 ms
 total time: 508.110046 ms
```

Output

When `quick` mode is disabled, the application performs the CPU reference implementation, which is used to validate the results from the GPU implementation by comparing the predicted latitudes and longitudes of each vertex with the CPU reference implementation. Further correctness checking was performed by comparing results to the HIVE reference implementation.

Geolocation application also supports the `quiet` mode, which allows the user to skip the output and just report the performance metrics (note, this will run the CPU implementation in the background without any output).

Performance and Analysis

Runtime is the key metric for measuring performance for Geolocation. We also check for prediction accuracy of the labels, but that is a threshold for correctness. If a certain threshold is not met (while comparing results to the CPU reference code), the output is considered incorrect and that run is invalid. Therefore, for the report we just focus on runtime.

Implementation limitations

Geolocation is also one of the few applications that exhibits a dynamic parallelism pattern:

- Parallel compute across the nodes,
- Serial compute per node, and
- Parallel compute within the serial compute per node.

One way to implement this will use the `ForAll()` operator for the parallel compute across the nodes, a simple while loop for the serial compute per node, and finally multiple `neighbor_reduce()` operators for the parallel work within the serial while loop. Currently, we do not have a way to support this within

Gunrock, but moving forward we can potentially leverage kernel launch within a kernel ("dynamic parallelism") to address this limitation.

Comparison against existing implementations

GPU	Dataset	|V|	|E|	Iterations	Spatial Iters	GTUSC (16 threads)	Gunrock (CPU)	Gunrock (GPU)
P100	sample	39	170	10	1000	N/A	0.144005 ms	0.022888 ms
P100	instagram	2373199	52711740	10	1000	8009.491 ms	1589.884031 ms	35.113831 ms
V100	twitter	50190344	880786012	10	1000	N/A	9216.666016 ms	66.108007 ms

On a workload that fills the GPU, gunrock outperforms GT's OpenMP C++ implementation by ~533x. Comparing gunrock's GPU vs. CPU performance, we see that gunrock's GPU version outperforms the CPU implementation by 100x. There is a lack of available datasets against which we can compare performance, so we use only the provided instagram and twitter datasets, and a toy sample for a sanity check on NVIDIA's P100 with 16GB of global memory and V100 with 32GB of global memory. All tested implementations meet the criteria of accuracy, which is validated against the output of the original python implementation.

- HIVE reference implementation uses distributed PySpark.
- GTUSC implementation uses C++ OpenMP.

Performance limitations

As discussed later in the "Alternate approaches" section, the current implementation of geolocation uses a compute operator with minimal load balancing. In cases where the graph is not so nicely distributed (where there is a great deal of difference in the degrees of vertices), the entire application will suffer significantly from load imbalance.

Profiling the application shows 98.78% of the compute time in GPU activities is in the `spatial_median` kernel, which gives us a good direction to focus our efforts on load-balancing the workloads within the operator. Specifically, we must target the `for` loops iterating over the neighbor list for spatial center calculations.

Next Steps

Alternate approaches

- **Neighborhood Reduce w/ Spatial Center:** We can perform better load balancing by leveraging a neighbor-reduce (`advance` operator + `cub::DeviceSegmentedReduce`) instead of using a compute operator. In graphs where the degrees of nodes vary a lot, the compute operator will be significantly slower than a load-balanced advance + segmented reduce.
- **Push Based Approach:** Instead of gathering all the locations from all the neighbors of an active vertex, we could instead perform a scatter of valid locations of all active vertices to their neighbors; this is a push approach vs. our current implementation's pull. Similar to the global gather approach, a push-based geolocation could also suffer from load imbalance, where some vertices will have to broadcast their valid locations to a long list of neighbors, while others will only have few neighbors to update. A push-based approach will also require a device synchronize before the spatial center computation, but may perform better by using an `advance_op` with an atomic update (note, pull is done using a `ForAll()`).

Gunrock implications

- **The `predicted` atomic:** Geolocation and some other applications exhibit the same behavior where the algorithm stops when all vertices' labels are predicted or determined. In Geolocation's case, when a location for all nodes is predicted, geolocation converges. We currently implement this with a loop and an atomic. This needs to be more of a core operation (mini-operator) such that when `isValidCount(labels|V|) == |V|`, a stop condition is met. Currently, we sidestep this issue by using a number-of-iterations parameter to determine the stop condition.
- **Parallel -> Serial -> Parallel:** As discussed earlier, gunrock currently doesn't have a way to address the dynamic parallelism problem, or even a kernel launch within a kernel. In geolocation's case, these minor parallel work inside the serial loop need to be multiple neighbor reduce.

Notes on multi-GPU parallelization

The challenging part for a multi-GPU Geolocation would be to obtain the updated node location from a separate device if the two vertices on different devices share an edge. An interesting approach here would be leveraging the P2P memory bandwidth with the new NVLink connectors to exchange a small amount of updates across the NVLink's memory lane; other ways are simply using direct accesses or explicit data movement. This is detailed more in the scaling documentation, but the communication model for multi-gpu geolocation could be done in the following way:

```
do
    Local geo location updates on local vertices;
    Broadcast local vertices' updates;
while no more update.
```

Notes on dynamic graphs

Streaming graphs is an interesting problem for the Geolocation application, because when predicting the location of a certain node, if another edge is introduced, the location of the vertex has to be recomputed entirely. This can still be done in an iterative manner, where if a node was inserted as a neighbor to a vertex, that vertex's predicted location will be marked invalid and during the next iteration it will be computed again along with all the other invalid vertices (locations).

Notes on larger datasets

If the datasets are larger than a single or multi-GPU's aggregate memory, the straightforward solution would be to let Unified Virtual Memory (UVM) in CUDA automatically handle memory movement.

Notes on other pieces of this workload

Geolocation calls a lot of CUDA math functions (`sin`, `cos`, `atan`, `atan2`, `median`, `mean`, `fminf`, `fmaxf`, etc.). Some of these micro-workloads can also leverage the GPU's parallelism; for example, a mean could be implemented using `reduce-mean/sum`. We currently don't have these math operators exposed within Gunrock in such a way they can be used in graph applications.

Research Potential

Further research is required to study Geolocation's dynamic parallelism pattern, it's memory access behavior, compute resource utilization, implementation details (API and core) and load balancing startegies for dynamic parallelism on the GPUs. Studying and understanding this pattern can allow us to create a more generalized approach for load balancing `parallel -> serial -> parallel` type of problems. It further invokes the question of studying when dynamic parallelism is better than mapping an alogirthm to a more conventional static approach (if possible).

Chapter 4

GraphSearch

The graph search (GS) workflow a walk-based method that searches a graph for nodes that score highly on some arbitrary indicator of interest.

The use case given by the HIVE government partner was sampling a graph: given some seed nodes, and some model that can score a node as "interesting", find lots of "interesting" nodes as quickly as possible. Their algorithm attempts to solve this problem by implementing several different strategies for walking the graph.

- `uniform`: given a node `u`, randomly move to one of `u`'s neighbors (ignoring scores)
- `greedy`: given a node `u`, walk to neighbor with maximum score
- `stochastic_greedy`: given a node `u`, choose neighbor to walk to with probability proportional to score

Use of these walk-based methods is motivated by the presence of homophily in many real world social networks: we expect interesting people to have relationships with interesting people.

Summary of Results

Graph search is relatively minor modification to Gunrock's random walk application, and was straightforward to implement. Though random walks are a "worst case scenario" for GPU memory bandwidth, we still achieve 3-5x speedup over a modified version of the OpenMP reference implementation.

The original OpenMP reference implementation actually ran slower with more threads – we fixed the bugs, but the benchmarking experience highlights the need for performant and hardened CPU baselines.

Until recently, Gunrock did not support parallelism *within* the lambda functions run by the `advance` operator, so neighbor selection for a given step in the walk

is done sequentially. Methods for exposing more parallism to the programmer are currently being developed via parallel neighbor reduce functions.

In an end-to-end graph search application, we'd need to implement the scoring function as well as the graph walk component. For performance, we'd likely want to implement the scoring function on the GPU as well, which makes this a good example of a "Gunrock+X" app, where we'd need to integrate the high-performance graph processing component with arbitrary user code.

Summary of Gunrock Implementation

The scoring model can be an arbitrary function (eg of node metadata). For example, if we were running GS on the Twitter friends/followers graph, the scoring model might be the output of a text classifier on each users' messages. Thus, we do not implement the scoring model in our Gunrock implementation – instead, we read scores from an input file and access them as necessary.

GS is a generalization of a random walk implementation, where there can be more variety in the transition function between nodes.

The GS `uniform` mode is exactly a uniform random walk, so we can use the pre-existing Gunrock application. Given a node, we compute the node to walk to as:

```
r = random.uniform(0, 1)
neighbors = graph.get_neighbors(node)
next_node = neighbors[floor(r * len(neighbors))]
```

Both the `GraphSearch` `greedy` and `stochastic_greedy` consist of small modifications to this transition function.

For `greedy`, we find the neighbor with maximum score:

```
neighbors = graph.get_neighbors(node)
next_node = neighbors[0]
next_node_score = scores[next_node]
for neighbor in neighbors:
    neighbor_score = scores[neighbor]
    if neighbor_score > next_node_score:
        next_node = neighbor
        next_node_score = neighbor_score
```

For `stochastic_greedy`, we sample neighbors proportional to their score – eg:

```
sum_neighbor_scores = 0
for neighbor in graph.neighbors(node):
```

```
   sum_neighbor_scores += scores[neighbor]

r *= sum_neighbor_scores

tmp = 0
for neighbor in graph.neighbors(node):
   tmp += scores[neighbor]
   if r < tmp:
       next_node = neighbor
       break
```

In Gunrock, we create a frontier containing all of the nodes we want to walk from. Then we map the transition function over the frontier using Gunrock's `ForEach` operator. Current nodes in the frontier are replaced with the chosen neighbor, and the walk is (optionally) recorded in an output array.

Because this is such a straightforward modification, we implement GS inside of the existing random walk `rw` Gunrock application. GS just requires adding a couple of extra flags and one extra array of size `|V|` to store the node values.

How To Run This Application on DARPA's DGX-1

Prereqs/input

```
git clone --recursive https://github.com/gunrock/gunrock -b dev-refactor
cd gunrock/tests/rw/
cp ../../gunrock/util/gitsha1.c.in ../../gunrock/util/gitsha1.c
make clean
make
```

Running the application

Application specific parameters

```
  --walk-mode
      0 = uniform
      1 = greedy
      2 = stochastic_greedy
  --node-value-path
      If --walk-mode != 0, this is the path to node scores
  --store-walks
      0 = just do the walk -- don't actually store it anywhere
      1 = store walks in memory
  --walk-length
      Length of each walk
  --walks-per-node
      Number of walks to do per seed node
```

```
  --seed
      Seed for random number generator
```

Example Command

```
# generate random features
python random-values.py 39 > chesapeake.values

# uniform random
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file ../../dataset/small/chesapeake.mt

# greedy
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file ../../dataset/small/chesapeake.mt

# stochastic greedy
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file ../../dataset/small/chesapeake.mt
```

Example Output

```
# --------------------------------------------
# uniform random

Loading Matrix-market coordinate-formatted graph ...
Reading from ../../dataset/small/chesapeake.mtx:
  Parsing MARKET COO format
 (39 nodes, 340 directed edges)...
Done parsing (0 s).
  Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...
Done converting (0s).
---------------------------
---------------------------
 Elapsed: 0.001907
Using advance mode LB
Using filter mode CULL
num_nodes=39
---------------------------
0    0   0   queue3      oversize :  234 ->  682
0    0   0   queue3      oversize :  234 ->  682
0    1   0   queue3      oversize :  682 ->  1085
0    1   0   queue3      oversize :  682 ->  1085
0    5   0   queue3      oversize :  1085 ->     1166
0    5   0   queue3      oversize :  1085 ->     1166
---------------------------
Run 0 elapsed: 4.551888, #iterations = 10
[[0, 38, 8, 35, 11, 25, 13, 27, 37, 7, ],
[1, 34, 1, 38, 30, 38, 29, 37, 7, 37, ],
```

```
[2, 17, 2, 38, 4, 38, 10, 18, 14, 28, ],
...
[36, 33, 0, 22, 38, 27, 37, 18, 38, 8, ],
[37, 21, 31, 17, 25, 17, 18, 32, 37, 26, ],
[38, 7, 8, 34, 6, 5, 6, 5, 38, 19, ]]
-------- NO VALIDATION -----[rw] finished.
 avg. elapsed: 4.551888 ms
 iterations: 10
 min. elapsed: 4.551888 ms
 max. elapsed: 4.551888 ms
 load time: 60.925 ms
 preprocess time: 964.890000 ms
 postprocess time: 0.715017 ms
 total time: 970.350027 ms

# ------------------------------------------------
# greedy
# !! In this case, the output is formatted as `GPU_result:CPU_result`, for correctness chec

Loading Matrix-market coordinate-formatted graph ...
Reading from ../../dataset/small/chesapeake.mtx:
  Parsing MARKET COO format
 (39 nodes, 340 directed edges)...
Done parsing (0 s).
  Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...
Done converting (0s).
--------------------------
--------------------------
 Elapsed: 0.085831
Using advance mode LB
Using filter mode CULL
num_nodes=39
--------------------------
0    0   0   queue3      oversize :  234 ->  682
0    0   0   queue3      oversize :  234 ->  682
0    1   0   queue3      oversize :  682 ->  770
0    1   0   queue3      oversize :  682 ->  770
--------------------------
Run 0 elapsed: 0.695944, #iterations = 10
[[0:0, 22:22, 32:32, 18:18, 11:11, 18:18, 11:11, 18:18, 11:11, 18:18, ],
[1:1, 22:22, 32:32, 18:18, 11:11, 18:18, 11:11, 18:18, 11:11, 18:18, ],
[2:2, 17:17, 2:2, 17:17, 2:2, 17:17, 2:2, 17:17, 2:2, 17:17, ],
...
[36:36, 33:33, 36:36, 33:33, 36:36, 33:33, 36:36, 33:33, 36:36, 33:33, ],
[37:37, 18:18, 11:11, 18:18, 11:11, 18:18, 11:11, 18:18, 11:11, 18:18, ],
[38:38, 2:2, 17:17, 2:2, 17:17, 2:2, 17:17, 2:2, 17:17, 2:2, ]]
```

```
0 errors occurred.
[rw] finished.
 avg. elapsed: 0.695944 ms
 iterations: 10
 min. elapsed: 0.695944 ms
 max. elapsed: 0.695944 ms
 load time: 44.2419 ms
 preprocess time: 974.721000 ms
 postprocess time: 0.731945 ms
 total time: 976.338863 ms

# -----------------------------------------------
# stochastic_greedy
# Output same format as `uniform` above.
# No correctness checking is implemented due to stochasticity.
```

Expected Output

When run in `--verbose` mode, the app outputs the walks. When run in `--quiet` mode, it outputs performance statistics (eg, total number of steps taken). If running `greedy GraphSearch`, the app also outputs the results of a correctness check. Correctness checks for `uniform` and `stochastic_greedy` are omitted because of their inherent stochasticity.

Validation

The correctness of the implementation has been validated in outside experiments, by making sure that the output walks are valid and the distribution of transitions is as expected.

Performance and Analysis

Performance is measured by the runtime of the app, given

- an input graph `G=(U, E)`
- set of seed nodes (hardcoded to all nodes in `G`)
- number of walks per seed
- number of steps per walk
- a transition function (eg `uniform|greedy|stochastic_greedy`)

Implementation limitations

The output of the random walk is a dense array of size `(# seeds) * (steps per walk) * (walks per seed)`. When we have a large graph *or* long walks *or* multiple walks per seed, this array may exceed the size of GPU memory.

At the moment, we only support walks starting from *all* of the nodes in `G`. It would be straightforward to add a parameter that would allow the use to specify a smaller set of seed nodes.

This app can only be used for graphs that have scores associated w/ each node. In order to run benchmarks, if scores are not available we often assign uniformly random scores to nodes. The distribution of these scores may effect the runtime of the algorithm by changing data access patterns – we test on the provided Twitter dataset, but do not have a variety of other node attributed graphs to test on.

Comparison against existing implementations

We measure runtime on the HIVE graphsearch Twitter dataset. This graph has `|U|=9291392` nodes and `|E|=21741663` edges.

At a high level, the results show:

Variant	OpenMP w/ 64 threads	Gunrock GPU	Gunrock Speedup
Directed greedy	236ms	**64ms**	3.7x
Directed random	158ms	**34ms**	4.6x
Undirected random	3186ms	**630ms**	5.0x

The undirected random walks take ~ 10x longer because directed walks terminate when they encounter a node without any neighbors and thus have average length significantly shorter than the `--walk-length` parameter.

Details and raw data follow.

HIVE Python reference implementation

We run the HIVE Python reference implementation w/ the following settings:

- undirected graph
- uniform transition function
- 1000 random seeds
- 128 steps per walk

With the `uniform` transition function, the run took 41 seconds. Walks are done sequentially, so runtime will scale linearly with the number of seeds. This implementation is *substantially* slower than even a single-threaded run of PNNLs OpenMP code. Thus, we omit further analysis.

PNNL OpenMP implementation

We run the PNNL OpenMP implementation on the Twitter graph w/ the following settings:

- commit: 69864383f0fc0e8aace52be34b329a2f8a58afb6
- 1,2,4,8,16,32 or 64 threads
- `greedy` or `uniform` transition function
- directed or undirected graph

We omit the `greedy` undirected case because the algorithm gets stuck jumping between a local maximum and it's highest scoring neighbor.

threads	method	directed?	nseeds	elapsed_sec	nsteps	steps_per_sec
1	greedy	yes	7199978	3.02876	16325873	5.39e+06
2	greedy	yes	7199978	2.83467	16325873	5.75e+06
4	greedy	yes	7199978	1.64405	16325873	9.93e+06
8	greedy	yes	7199978	0.870028	16325873	1.87e+07
16	greedy	yes	7199978	0.605769	16325873	2.69e+07
32	greedy	yes	7199978	0.43742	16325873	3.73e+07
64	greedy	yes	7199978	**0.236701**	16325873	6.89e+07
1	unif.	yes	7199978	**14.6291**	14510781	**991915**
2	unif.	yes	7199978	24.2175	14186833	585809
4	unif.	yes	7199978	25.1764	14487202	575427
8	unif.	yes	7199978	27.7312	13937449	502591
16	unif.	yes	7199978	30.5377	14062226	460488
32	unif.	yes	7199978	32.1057	13906144	433137
64	unif.	yes	7199978	31.2754	13876284	443680
1	unif.	no	100000	**12.3982**	12700000	**1.024+06**
2	unif.	no	100000	19.7925	12700000	641658
4	unif.	no	100000	22.5432	12700000	563362
8	unif.	no	100000	26.1053	12700000	486491
16	unif.	no	100000	28.275	12700000	449160
32	unif.	no	100000	28.334	12700000	448224
64	unif.	no	100000	28.7419	12700000	441864

Note that we use fewer seesd for the undirected uniform case due to slow runtime.

Observe that the `rand` modes have very bad scaling as a function of cores. After investigation, this was due to two issues. First, the neighbors were being sampled incorrectly, which led to chaotic behavior. Second, the app was using a slow random number generator w/ an excessive number of seed resets. We created a PR to fix those issues here.

After these fixes, runtimes were as follows:

threads	method	directed?	nseeds	elapsed_sec	nsteps	steps_per_sec
1	greedy	yes	7199978	3.02876	16325873	5.39e+06

threads	method	directed?	nseeds	elapsed_sec	nsteps	steps_per_sec
2	greedy	yes	7199978	2.83467	16325873	5.75e+06
4	greedy	yes	7199978	1.64405	16325873	9.93e+06
8	greedy	yes	7199978	0.870028	16325873	1.87e+07
16	greedy	yes	7199978	0.605769	16325873	2.69e+07
32	greedy	yes	7199978	0.43742	16325873	3.73e+07
64	greedy	yes	7199978	**0.236701**	16325873	**6.89e+07**
1	unif.	yes	7199978	1.49886	16529694	1.10e+07
2	unif.	yes	7199978	1.60176	16533004	1.03e+07
4	unif.	yes	7199978	0.974128	16538957	1.69e+07
8	unif.	yes	7199978	0.455227	16534756	3.63+07
16	unif.	yes	7199978	0.257524	16528617	6.42e+07
32	unif.	yes	7199978	**0.155722**	13906144	**1.06e+08**
64	unif.	yes	7199978	0.158828	16537488	1.04e+08
1	unif.	no	7199978	125.963	914397206	1.92e+08
2	unif.	no	7199978	78.927	914397206	1.80e+08
4	unif.	no	7199978	39.7097	914397206	2.96e+08
8	unif.	no	7199978	22.5195	914397206	6.35e+08
16	unif.	no	7199978	11.0047	914397206	1.12e+09
32	unif.	no	7199978	5.56317	914397206	**1.85e+09**
64	unif.	no	7199978	**3.18615**	914397206	1.82e+09

Note the improved runtimes and scaling. These experiments were run with this branch at commit 6c25a0687eecebfd4393e86fa4c7308d5594b73d.

All experiments conducted on the HIVE DGX-1.

Gunrock GPU implementation

directed, greedy

```
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file dir_gs_twitter.mtx \
    --node-value-path gs_twitter.values \
    --walk-mode 1 \
    --walk-length 32 \
    --undirected=0 \
    --store-walks 0 \
    --quick \
    --num-runs 10


Loading Matrix-market coordinate-formatted graph ...
Reading from dir_gs_twitter.mtx:
  Parsing MARKET COO format
 (7199978 nodes, 21741663 directed edges)...
```

```
Done parsing (7 s).
  Converting 7199978 vertices, 21741663 directed edges ( ordered tuples) to CSR format...
Done converting (0s).
================================================
 advance-mode=LB
Using advance mode LB
Using filter mode CULL
Run 0 elapsed: 65.273046, #iterations = 32
Run 1 elapsed: 64.157963, #iterations = 32
Run 2 elapsed: 64.009190, #iterations = 32
Run 3 elapsed: 64.055920, #iterations = 32
Run 4 elapsed: 64.069033, #iterations = 32
Run 5 elapsed: 64.002037, #iterations = 32
Run 6 elapsed: 64.031839, #iterations = 32
Run 7 elapsed: 64.036846, #iterations = 32
Run 8 elapsed: 64.065933, #iterations = 32
Run 9 elapsed: 64.047098, #iterations = 32
Validate_Results: total_neighbors_seen=298668024
Validate_Results: total_steps_taken=16325873
-------- NO VALIDATION --------
[rw] finished.
 avg. elapsed: 64.174891 ms
 iterations: 32
 min. elapsed: 64.002037 ms
 max. elapsed: 65.273046 ms
 load time: 7086.91 ms
 preprocess time: 1016.620000 ms
 postprocess time: 101.121902 ms
 total time: 2073.837996 ms
```

directed, uniform

```
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file dir_gs_twitter.mtx \
    --node-value-path gs_twitter.values \
    --walk-mode 0 \
    --walk-length 128 \
    --undirected=0 \
    --store-walks 0 \
    --quick \
    --num-runs 10 \
    --seed 123

Loading Matrix-market coordinate-formatted graph ...
Reading from dir_gs_twitter.mtx:
  Parsing MARKET COO format
```

```
 (7199978 nodes, 21741663 directed edges)...
Done parsing (7 s).
  Converting 7199978 vertices, 21741663 directed edges ( ordered tuples) to CSR format...
Done converting (1s).
============================================
 advance-mode=LB
Using advance mode LB
Using filter mode CULL
------------------------
Run 0 elapsed: 38.613081, #iterations = 128
Run 1 elapsed: 34.458876, #iterations = 128
Run 2 elapsed: 34.530163, #iterations = 128
Run 3 elapsed: 33.849001, #iterations = 128
Run 4 elapsed: 33.759117, #iterations = 128
Run 5 elapsed: 33.967972, #iterations = 128
Run 6 elapsed: 33.873081, #iterations = 128
Run 7 elapsed: 33.970118, #iterations = 128
Run 8 elapsed: 33.756971, #iterations = 128
------------------------
Run 9 elapsed: 33.715963, #iterations = 128
Validate_Results: total_neighbors_seen=289124779
Validate_Results: total_steps_taken=16530404
-------- NO VALIDATION --------
[rw] finished.
 avg. elapsed: 34.449434 ms
 iterations: 128
 min. elapsed: 33.715963 ms
 max. elapsed: 38.613081 ms
 load time: 7176.17 ms
 preprocess time: 1016.720000 ms
 postprocess time: 101.902962 ms
 total time: 1781.071901 ms
```

undirected, uniform

```
./bin/test_rw_9.1_x86_64 --graph-type market --graph-file undir_gs_twitter.mtx \
    --node-value-path gs_twitter.values \
    --walk-mode 0 \
    --walk-length 128 \
    --store-walks 0 \
    --quick \
    --num-runs 10 \
    --seed 123

Loading Matrix-market coordinate-formatted graph ...
```

```
Reading from undir_gs_twitter.mtx:
  Parsing MARKET COO format
 (7199978 nodes, 43483326 directed edges)...
Done parsing (7 s).
  Converting 7199978 vertices, 43483326 directed edges ( ordered tuples) to CSR format...
Done converting (0s).
==============================================
 advance-mode=LB
Using advance mode LB
Using filter mode CULL
Run 0 elapsed: 636.021852, #iterations = 128
Run 1 elapsed: 631.129026, #iterations = 128
Run 2 elapsed: 631.053925, #iterations = 128
Run 3 elapsed: 631.713152, #iterations = 128
Run 4 elapsed: 631.028175, #iterations = 128
Run 5 elapsed: 631.374836, #iterations = 128
Run 6 elapsed: 631.196976, #iterations = 128
Run 7 elapsed: 632.030964, #iterations = 128
Run 8 elapsed: 631.026983, #iterations = 128
Run 9 elapsed: 630.996943, #iterations = 128
Validate_Results: total_neighbors_seen=75443835041
Validate_Results: total_steps_taken=914397206
-------- NO VALIDATION --------
[rw] finished.
 avg. elapsed: 631.757283 ms
 iterations: 128
 min. elapsed: 630.996943 ms
 max. elapsed: 636.021852 ms
 load time: 7705.9 ms
 preprocess time: 1010.830000 ms
 postprocess time: 102.057934 ms
 total time: 7755.448818 ms
```

Performance limitations

- For the undirected uniform settings, profiling shows that 79% of compute time is spent in the `ForAll` operator and 20% is spent in the `curand` random number generator. Device memory bandwidth in the `ForAll` kernel is 193GB/s.
- For the directed greedy settings, profiling shows that 99.5% of compute time is spent in the `ForAll` operator. Device memory bandwidth in the `ForAll` kernel is 136GB/s.

When we do a large number of walks and/or the length of each walk is very long, there may not be enough GPU memory to store all of the walks in memory. For now, we expose the `--store-walks` parameter – when this is set to zero, the

walk is discarded as it is computed and only the length of the walk is stored. A better solution that could be implemented in the future would be to move walks from GPU to CPU memory as they grow too large.

Optimization: In a directed walk, once we hit a node with no outgoing neighbors, we halt the walk. In the current Gunrock implementation, the enactor runs for a fixed number of iterations, regardless of whether any of the nodes are still active. It would be straightforward to add a check that terminates the app when no "living" nodes are left.

Next Steps

Alternate approaches

The size of the output array may become a significant bottleneck for large graphs. However, since all of the transition functions do not depend on anything besides the current node, we could reasonably move the results of the walk from GPU to CPU memory every N iterations. Properly executed, this should eliminate the largest bottleneck without unduely impacting performance.

Gunrock implications

For the `greedy` and `stochastic_greedy` transition function, we have to sequentially iterate over all of a node's neighbors. Simple wrappers for computing eg. the maximum of node scores across all of a nodes neighbors could be helpful, both for ease of programming and performance. Gunrock has a newly added `NeighborReduce` kernel that supports associative reductions – it should be straightforward to implement (at least) the `greedy` transition function with this kernel. The `stochastic_greedy` transition function would require a more complex reduction function along the lines of reservoir sampling.

Notes on multi-GPU parallelization

If the graph is small enough to be duplicated on each GPU, the implementation is trivial: just do a subset of the walks on each GPU. The scalability will be perfect, as there is no communication involved at all.

When the graph is distributed across multiple GPUs, we expect to have very poor scalability, as the ratio of computation to communication is very low. A more detailed discussion is available here.

Notes on dynamic graphs

This workflow does not have an explicit dynamic component. However, because steps only depend on the current node, the underlying graph could change as the walks.

Notes on larger datasets

The random accesses inherent to graph search make it a particularly difficult workflow for larger than GPU memory datasets. The most straightforward solution would be to let Unified Virtual Memory (UVM) in CUDA automatically handle memory movement, but we should expect to see a substantial reduction in performance.

Notes on other pieces of this workload

In real use cases, the scoring function would be computed lazily – that is, we wouldn't have a precomputed array with scores for each of the nodes, and we would need to run the scoring function as the walk is running. Thus, it would be critical for us to be able to call the scoring function from within Gunrock quickly and without excessive programmer overhead.

Chapter 5

Local Graph Clustering (LGC)

From Andersen et al.:

> A local graph partitioning algorithm finds a cut near a specified starting vertex, with a running time that depends largely on the size of the small side of the cut, rather than the size of the input graph.

A common algorithm for local graph clustering is called PageRank-Nibble (PRNibble), which solves the L1 regularized PageRank problem. We implement a coordinate descent variant of this algorithm found in Fountoulakis et al., which uses the fast iterative shrinkage-thresholding algorithm (FISTA).

Summary of Results

This variant of local graph clustering (L1 regularized PageRank via FISTA) is a natural fit for Gunrock's frontier-based programming paradigm. We observe speedups of 2-3 orders of magnitude over the HIVE reference implementation.

The reference implementation of the algorithm was not explicitly written as `advance`/`filter`/`compute` operations, but we were able to quickly determine how to map the operations by using a lightweight Python implementation of the Gunrock programming API as a development environment. Thus, LGC was a good exercise in implementing a non-trivial end-to-end application in Gunrock from scratch.

Summary of Gunrock Implementation

We implement Algorithm 2 from Fountoulakis et al., which maps nicely to Gunrock. We present the pseudocode below along with the corresponding Gunrock operations:

```
A: adjacency matrix of graph
D: diagonal degree matrix of graph
Q: D^(-1/2) x (D - (1 - alpha)/2 x (D + A)) x D^(-1/2)
s: teleportation distribution, a distribution over nodes of graph
d_i: degree of node i
p_0: PageRank vector at iteration 0
q_0:  D^(-1/2) x p term that coordinate descent optimizes over
f(q): 1/2<q, Qq> - alpha x <s, D^(-1/2) x q>
grad_f_i(q_0): i'th term of the gradient of f(q_0) using q at iteration 0
rho: constant used to ensure convergence
alpha: teleportation constant in (0, 1)

Initialize: rho > 0
Initialize: q_0 = [0 ... 0]
Initialize: grad_f(q_0) = -alpha x D^(-1/2) x s

For k = 0, 1, ..., inf
    // Implemented using Gunrock ForAll operator
    Choose an i such that grad_f_i(q_k) < - alpha x rho x d_i^(1/2)
    q_k+1(i) = q_k(i) - grad_f_i(q_k)
    grad_f_i(q_k+1) = (1 - alpha)/2 x grad_f_i(q_k)

    // Implemented using Gunrock Advance and Filter operator
    For each j such that j ~ i
        Set grad_f_j(q_k+1) = grad_f_j(q_k) +
            (1 - alpha)/(2d_i^(1/2) x d_j^(1/2)) x A_ij x grad_f_i(q_k)

    For each j such that j !~ j
        Set grad_f_j(q_k+1) = grad_f_j(q_k)

    // Implemented using Gunrock ForEach operator
    // Note: ||y||_inf is the infinity norm
    if (||D^(-1/2) x grad_f(q_k)||_inf > rho x alpha)
            break
EndFor

return p_k = D^(1/2) x q_k
```

How To Run This Application on DARPA's DGX-1

Prereqs/input

```
# clone gunrock
git clone --recursive https://github.com/gunrock/gunrock.git \
        -b dev-refactor
```

```
cd gunrock/tests/pr_nibble
cp ../../gunrock/util/gitsha1.c.in ../../gunrock/util/gitsha1.c
make clean
make
```

Running the application

Example command

```
./bin/test_pr_nibble_9.1_x86_64 \
    --graph-type market \
    --graph-file ../../dataset/small/chesapeake.mtx \
    --src 0 \
    --max-iter 1
```

Example output

```
Loading Matrix-market coordinate-formatted graph ...
  Reading meta data from ../../dataset/small/chesapeake.mtx.meta
  Reading edge lists from ../../dataset/small/chesapeake.mtx.coo_edge_pairs
  Substracting 1 from node Ids...
  Edge doubleing: 170 -> 340 edges
  graph loaded as COO in 0.084587s.
Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...Done (0s).
Degree Histogram (39 vertices, 340 edges):
    Degree 0: 0 (0.000000 %)
    Degree 2^0: 0 (0.000000 %)
    Degree 2^1: 1 (2.564103 %)
    Degree 2^2: 22 (56.410256 %)
    Degree 2^3: 13 (33.333333 %)
    Degree 2^4: 2 (5.128205 %)
    Degree 2^5: 1 (2.564103 %)


---------------------------
pr_nibble::CPU_Reference: reached max iterations. breaking at it=10
---------------------------
 Elapsed: 0.103951
============================================
 advance-mode=LB
Using advance mode LB
Using filter mode CULL
---------------------------
0    2   0   queue3      oversize :  234 ->  342
0    2   0   queue3      oversize :  234 ->  342
pr_nibble::Stop_Condition: reached max iterations. breaking at it=10
---------------------------
Run 0 elapsed: 1.738071, #iterations = 10
```

```
0 errors occurred.
[pr_nibble] finished.
 avg. elapsed: 1.738071 ms
 iterations: 140733299213840
 min. elapsed: 1.738071 ms
 max. elapsed: 1.738071 ms
 src: 0
 nodes_visited: 41513344
 edges_visited: 140733299212960
 nodes queued: 140733299212992
 edges queued: 5424992
 load time: 116.627 ms
 preprocess time: 963.004000 ms
 postprocess time: 0.080824 ms
 total time: 965.005875 ms
```

Expected Output

We do not print the actual output values of PRNibble, but we output the results of a correctness check of the GPU version against our CPU implementation. `0 errors occurred.` indicates that LGC has generated an output that exactly matches our CPU validation implementation.

Our implementations are validated against the HIVE reference implementation.

For ease of exposition, and to help in mapping the workflow to Gunrock primitives, we also implemented a version of PRNibble in pygunrock. This implementation is nearly identical to the actual Gunrock app, but in a way that more clearly exposes the logic of the app and eliminates a lot of Gunrock scaffolding/memory management/etc.

Performance and Analysis

Performance is measured by the runtime of the approximate PageRank solver, given

- a graph `G=(U, E)`
- a (set of) seed node(s) `S`
- some parameters controlling e.g., the target conductivity of the output cluster (`rho`, `alpha`, ...)

The reference implementation also includes a sweep-cut step, where a threshold is applied to the approximate PageRank values to produce hard cluster assignments. We do not implement this part of the workflow, as it is not fundamentally a graph operation.

Implementation limitations

PageRank runs on arbitrary graphs – it does not require any special conditions such as node attributes, etc.

- **Memory size**: The dataset is assumed to be an undirected graph (with no self-loops). We were able to run on graphs of up to 6.2 GB in size (7M vertices, 194M edges). The memory limitation should be the number of edges `2*|E| + 7*|U|`, which needs to be smaller than the GPU memory size (16 GB for a single P100 on DGX-1).
- **Data type**: We have only tested our implementations using an `int32` data type for node IDs. However, we could also support `int64` node IDs for graphs with more than 4B edges.

Comparison against existing implementations

We compare our Gunrock GPU implementation with two CPU reference implementations:

- HIVE reference implementation (Python wrapper around C++ library)
- Gunrock CPU reference implementation (C++)

We find the Gunrock implementation is 3 orders of magnitude faster than either reference CPU implementation. The minimum, geometric mean, and maximum speedups are 7.25x, 1297x, 32899x, respectively.

All runtimes are in milliseconds (ms):

Dataset	HIVE Ref. C++	Gunrock C++	Gunrock GPU	Speedup
delaunay_n13	21.52	16.33	2.86	8
ak2010	97.99	72.08	3.04	32
coAuthorsDBLP	1004	1399	4.86	207
belgium_osm	2270	1663	2.97	726
roadNet-CA	3403	2475	3.03	1123
delaunay_n21	5733	4084	2.98	1924
cit-Patents	40574	22148	16.41	2472
hollywood-2009	43024	30430	46.30	929
road_usa	48232	31617	3.01	16024
delaunay_n24	49299	34655	3.28	15030
soc-LiveJournal1	63151	37936	19.29	3274
europe_osm	97022	72973	2.95	32889
indochina-2004	101877	71902	11.05	9220
kron_g500-logn21	110309	89438	627.55	176
soc-orkut	111391	89752	18.05	6171

Performance limitations

We profiled the Gunrock GPU primitives on the `kron_g500-logn21` graph. The profiler adds approx. 100 ms of overhead (728.48 ms with profiler vs. 627.55 ms without profiler). The breakdown of runtime by kernel looks like:

Gunrock Kernel	Runtime (ms)	Percentage of Runtime
Advance (EdgeMap)	566.76	77.8%
Filter (VertexMap)	10.85	1.49%
ForAll (VertexMap)	2.90	0.40%
Other	147.89	20.3%

Note: "Other" includes HtoD and DtoH memcpy, smaller kernels such as scan, reduce, etc.

By profiling the LB Advance kernel, we find that the performance of `advance` is bottlenecked by random memory accesses. In the first part of the computation – getting row pointers and column indices – memory accesses can be coalesced and the profilers says we perform 4.9 memory transactions per access, which is close to the ideal of 4. However, once we start processing these neighbors, the memory access becomes random and we perform 31.2 memory transactions per access.

Next Steps

Alternate approaches

PRNibble can also be implemented in terms of matrix operations using our GPU GraphBLAS library – this implementation is currently in progress.

In theory, local graph clustering is appealing because you don't have to "touch" the entire graph. However, all LGC implementations that we are aware of first load the entire graph into CPU/GPU memory, which limits the size of the graph that can be analyzed. Implementations that load data from disk "lazily" as computation happens would be interesting and practically useful.

Ligra includes some high performance implementations of similar algorithms. In the future, it would be informative to benchmark our GPU implementation against those performance optimized multi-threaded CPU implementations.

Gunrock implications

Gunrock currently supports all of the operations needed for this application. In particular, the `ForAll` and `ForEach` operators were very useful for this application.

Additionally, `pygunrock` proved to be a useful tool for development – correctly mapping the original (serial) algorithm to the Gunrock operators required a lot of attention to detail, and having an environment for rapid expedited experimentation facilitated the algorithmic development.

Notes on multi-GPU parallelization

Since this problem maps well to Gunrock operations, we expect parallelization strategy would be similar to BFS and SSSP. The dataset can be effectively divided across multiple GPUs.

Notes on dynamic graphs

It is not obvious how this algorithm would be extended to handle dynamic graphs. At a high level, the algorithm iteratively spreads mass from the seed nodes to neighbors in the graph. As the connectivity structure of the graph changes, the dynamics of this mass spreading could change arbitrarily – imagine edges that form bridges between two previously distinct clusters. Thus, we suspect that adapting the app to work on dynamic graphs may require substantial development and study of the underlying algorithms.

Notes on larger datasets

If the data were too big to fit into the aggregate GPU memory of multiple GPUs on a single node, then we would need to look at multiple-node solutions. Getting the application to work on multiple nodes would not be challenging, because it is very similar to BFS. However, optimizing it to achieve good scalability may require asynchronous communication, an area where we have some experience (Pan et al.). Asynchronous communication may be necessary in order to reach better scalability in multi-node, because this application which can be formulated as sparse-matrix vector multiplication has limited computational intensity (low computation-to-communication).

Notes on other pieces of this workload

As mentioned previously, we do not implement the sweep-cut portion of the workflow where the PageRank values are discretized to produce hard cluster assignments. Though it's not fundamentally a graph problem, parallelization of this step is a research question addressed in Shun et al.

Potential future academic work

The coordinate descent implementation (developed by Ben Johnson) shows that Gunrock can be used as a coordinate descent solver. There has been more interest in coordinate descent recently, because coordinate descent can be used in ML as an alternative to stochastic gradient descent for SVM training.

References

Prof. Cho-Jui Hsieh from UC Davis is an expert in this field (see 1, 2, 3).

Chapter 6

Graph Projections

Given a (directed) graph `G`, graph projection outputs a graph `H` such that `H` contains edge `(u, v)` iff `G` contains edges `(w, u)` and `(w, v)` for some node `w`. That is, graph projection creates a new graph where nodes are connected iff they are neighbors of the same node in the original graph. Typically, the edge weights of `H` are computed via some (simple) function of the corresponding edge weights of `G`.

Graph projection is most commonly used when the input graph `G` is bipartitite with node sets `U1` and `U2` and directed edges `(u, v)`. In this case, the operation yields a unipartite projection onto one of the node sets. However, graph projection can also be applied to arbitrary (unipartite) graphs.

Summary of Results

Because it has a natural representation in terms of sparse matrix operations, graph projections gave us an opportunity to compare ease of implementation and performance between Gunrock and another UC-Davis project, GPU GraphBLAS.

Overall, we found that Gunrock was more flexible and more performant than GraphBLAS, likely due to better load balancing. However, in this case, the GraphBLAS application was substantially easier to program than Gunrock, and also allowed us to take advantage of some more sophisticated memory allocation methods available in the GraphBLAS cuSPARSE backend. These findings suggest that addition of certain commonly used API functions to Gunrock could be a fruitful direction for further work.

Summary of Gunrock Implementation

We implement two versions of graph projections: one using Gunrock and one using GraphBLAS.

Gunrock

First, we can compute graph projection in Gunrock via a single `advance` operation from all nodes w/ nonzero outgoing degree:

```
def _advance_op(self, G, H_edges, src, dest):
    for neib in G.neighbors(src):
        if dest != neib:
            H_edges[dest * G.num_nodes + neib] += 1
```

That is, for each edge in the graph, we fetch the neighbors of the source node in `G`, then increment the weight of the edge between `dest` and each of those neighbors in `H_edges`.

Note that we have only implemented the unweighted case and a single method for computing the edgeweights of `H`, but the extension to weighted graphs and different weighting functions would be straightforward.

We use a dense `|V|x|V|` array to store the edges of the output matrix `H`. This is simple and fast, but uses an unreasonably large amount of memory (a graph with 60k nodes requires 16 GB). In the worst-case scenario, `H` may actually have all `|V|x|V|` possible edges, but any typical real-world graph has *far* fewer edges in practice.

GraphBLAS

Second, we implement graph projection as a single sparse matrix-matrix multiply in our GraphBLAS GPU library, which wraps and extends cuSPARSE.

Graph projection admits a simple linear algebra formulation. Given the adjacency matrix `A` of graph `G`, the projection is just:

```
H = matmul(transpose(A), A)
```

which can be concisely implemented via cuSPARSE's `csr2csc` and `csrgemm` functions.

The `csrgemm` functions in cuSPARSE allocate memory more intelligently than we do above, on the order of the number of edges in the output. Thus, our GraphBLAS implementation can scale to substantially larger matrices than our Gunrock implementation. However, implementing graph projection via a single call to `csrgemm` requires both the input graph `G` and output graph `H` to fit in GPU memory (16 GB on the DGX-1). This limit can easily be hit, even for a moderately sized `G`, as the number of edges in `H` is often orders of magnitude larger than in `G`.

Thus, to scale to larger graphs, we implement graph projections via a chunked matrix multiply. Specifically, to compute `matmul(X, Y)` w/ `X.shape = (n,`

`m)` and `Y.shape = (m, k)`, we split `X` into `c` matrices `(X_1, ..., X_c)`, w/ `X_i.shape = (n / c, m)`. Then we compute `matmul(X_i, Y)` for each `X_i`, moving the output of each multiplication from GPU to CPU memory as we go. This implementation addresses the common case where we can fit both `X` and `Y` in GPU memory, but not `matmul(X, Y)`. Obviously, the chunked matrix multiply incurs a performance penalty, but allows us to run graph projections of much larger graphs on the GPU.

How To Run This Application on DARPA's DGX-1

Gunrock

Prereqs/input

```
git clone --recursive https://github.com/gunrock/gunrock -b dev-refactor
cd gunrock/tests/proj/
cp ../../gunrock/util/gitsha1.c.in ../../gunrock/util/gitsha1.c
make clean
make
```

Application specific parameters

None

Example Command

```
./bin/test_proj_9.1_x86_64 \
        --graph-type market \
        --graph-file ../../dataset/small/chesapeake.mtx
```

Example Output

```
Loading Matrix-market coordinate-formatted graph ...
Reading from ../../dataset/small/chesapeake.mtx:
  Parsing MARKET COO format edge-value-seed = 1539110067
 (39 nodes, 340 directed edges)...
Done parsing (0 s).
  Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...
Done converting (0s).

--------------------------
--------------------------
 Elapsed: 0.026941
Using advance mode LB
Using filter mode CULL

--------------------------
0    0   0   queue3      oversize :  234 ->  342
0    0   0   queue3      oversize :  234 ->  342
```

```
--------------------------
Run 0 elapsed: 0.199080, #iterations = 1
edge_counter=1372
0->1 | GPU=9.000000 CPU=9.000000
0->2 | GPU=1.000000 CPU=1.000000
0->3 | GPU=2.000000 CPU=2.000000
...
38->35 | GPU=28.000000 CPU=28.000000
38->36 | GPU=2.000000 CPU=2.000000
38->37 | GPU=18.000000 CPU=18.000000
======= PASSED ======
[proj] finished.
 avg. elapsed: 0.199080 ms
 iterations: 38594739
 min. elapsed: 0.199080 ms
 max. elapsed: 0.199080 ms
 src: 0
 nodes_visited: 38578864
 edges_visited: 38578832
 nodes queued: 140734466796512
 edges queued: 140734466795232
 load time: 85.5711 ms
 preprocess time: 955.861000 ms
 postprocess time: 3.808022 ms
 total time: 960.005045 ms
```

Expected Output

When run in `verbose` mode, the app outputs the weighted edgelist of the graph projection `H`. When run in `quiet` mode, it only outputs performance statistics and the results of a correctness check.

GraphBLAS

Prereqs/input

```
git clone --recursive https://github.com/bkj/graphblas_proj
cd graphblas_proj
make clean
make
```

Application specific parameters

```
--unweighted
 1 = convert entries of adjacency matrix to 1
 0 = leave entries of adjacency matrix as-is
--proj-debug
```

```
 1 = print debug information
 0 = don't print debug information
--print-results
 1 = print the edges of the projected graph
 0 = don't print edges
--onto-cols
 1 = given adjacency matrix A w/ `A.shape = (m, n)`,
     compute projection `H = matmul(transpose(A), A)` w/ `H.shape = (n, n)`
 0 = given adjacency matrix A w/ `A.shape = (m, n)`,
     compute projection `H = matmul(A, transpose(A))` w/ `H.shape = (m, m)`
--num-chunks
 <= 1 = do matrix multiply in one step
  > 1 = break matrix multiply into multiple chunks (eg, so we can compute
        projections larger than GPU memory)
```

Example Command

```
# generate some random data `data/X.mtx`
python data/make-random.py --seed 111 \
        --num-rows 1000 --num-cols 1000 --density 0.1

# run graph projection
./proj --X data/X.mtx --unweighted 1 --proj-debug 1
```

Example Output

```
proj.cu: loading data/X.mtx
  done
proj.cu: computing transpose
  done
proj.cu: computing projection
mxm analyze successful!
mxm compute successful!
  done
proj_num_edges          = 999946  # number of edges in H, including self loops
dim_out                 = 1000    # dimension of projected graph
proj_num_edges (noloop) = 998946  # number of edges in H, excluding self loops
timer                   = 208.98  # elapsed time (no IO)
```

Expected Output

The app will print the number of edges in the projected graph. Additionally,

- When run w/ `--print-results=1`, the app prints the edges of the the graph projection `H`.
- When run w/ `--proj-debug=1`, the app prints a small number of progress messages.

Validation

We compared the results of the Gunrock implementation to the HIVE reference implementation and the PNNL implementation. These two implementations vary slightly in their output (e.g., handling of self loops). We validated the correctness of our results against the HIVE reference implementation.

Performance and Analysis

Performance is measured by the runtime of the app, given:

- an input graph `G` (possibly bipartite)
- whether to project onto the rows or columns of the graph.

Implementation limitations

Gunrock

The primary limitation of the current implementation is that it allocates a `|V|x|V|` array, where `|V|` is the number of nodes in the network. This means that the memory requirements of the app can easily exceed the memory available on a single GPU (16 GB on the DGX-1). The size of this array reflects the *worst case* memory requirements of the graph projection workflow; while some graphs can become exceptionally large and dense when projected, we should be able to run the app on larger graphs if we store the output in a different data structure and/or allocate memory more efficiently. Algorithms do exist for mitigating this issue: cuSPARSE's `csrgemm` method computes the row pointers in one pass, then allocates the exact amount of memory for `H`'s column and value arrays, then actually computes the matrix product. An interesting future direction would be to integrate this sort of algorithm into Gunrock.

It may be possible to improve performance by making assumptions about the topology of the graph. Graph projection is often used for bipartite graphs, but this app does not make any assumptions about the topology of the graph. This choice was made in order to remain consistent with the HIVE reference implementation.

There are various ways that the edges of the output graph `H` can be weighted. We only implement graph projections for unweighted graphs: the weight of the edge `(u, v)` in the output graph `H` is the count of (incoming) neighbors that `u` and `v` have in common in the original graph `G`. Implementation of other weight functions would be fairly straightforward.

GraphBLAS

Currently, for the chunked matrix multiply, the CPU memory allocation and GPU to CPU memory copies for `matmul(X_i, Y)` block the computation of

`matmul(X_[i+1], Y)`. We could implement this in a non-blocking way using CUDA streams, but this would require some redesign of the GraphBLAS APIs.

Certain weighting functions are easily implemented by applying a transformation to the values of the sparse adjacency matrix `A`, but others cannot. For instance, these weighting functions are easy to implement:

```
weight_out = 1                                (by setting both matrices entries to 1)
weight_out = weight_edge_1                    (by setting one matrix's entries to 1)
weight_out = weight_edge_1 / weight_edge_2 (by setting one matrix's entries to 1 / x)
...
```

while this function (from the HIVE reference implementation) is not easy to implement in the `cuSPARSE` plus/multiply semiring:

```
weight_out = weight_edge_1 / (weight_edge_1 + weight_edge_2)
```

Implementation of additional semirings in GraphBLAS is currently in progress, and will extend the family of weightings we could support.

Comparison against existing implementations

When the graph fits in its memory, the Gunrock implementation is approx. 5x faster than the GraphBLAS implementation and approx. 100x faster than PNNL's OpenMP CPU implemention w/ 64 threads. Somewhat surprisingly, PNNL's implementation is substantially slower than a single-threaded scipy sparse matrix multiplication.

When the graph does not fit in the Gunrock implementation's memory, our GPU GraphBLAS implementation is the fastest of the remaining implementations.

Existing implementations

PNNL

We compare our results against PNNL's OpenMP reference implementation. We make minor modifications to their code to handle unweighted graphs, in order to match the Gunrock and GraphBLAS implementations. That is, the weight of the edge in the output graph `H` is the number of shared neighbors in the original graph.

There is a `--simple` flag in PNNL's CLI, but examining the code reveals that it just changes the order of operations. Thus, all of our experiments are conducted with `--simple=1`, which is faster than `--simple=0` due to better data access patterns.

Scipy

A very simple baseline is sparse matrix-matrix multiplication as implemented in the popular `scipy` python package. This is a single-threaded C++ implementation with a Python wrapper. Note, this implementation comes with the same caveats about weighting functions as the Gunrock implementation.

Experiments

MovieLens

MovieLens is a bipartite graph `G=(U, V, E)` w/ `|U|=138493`, `|V|=26744` and `|E|=20000264`. We report results on the full graph, as well as several random subgraphs.

For PNNL's OpenMP implementation, we report results using {1, 2, 4, 8, 16, 32, 64} threads.

In all cases, we project onto the nodeset `|V|`, producing a `|V|x|V|` graph.

Note: Small differences in the number of nonzero entries in the output (nnz_out) are due to small book-keeping differences (specifically, keeping or dropping self-loops). These differences do not have any meaningful impact on runtimes.

1M edge subgraph (|U|=6743 |V|=13950 |E|=1M)

implementation	num_threads	nnz_out	elapsed_seconds
scipy	1	63104132	2.4912
PNNL OpenMP	1	63090182	61.4852
PNNL OpenMP	2	63090182	62.2842
PNNL OpenMP	4	63090182	60.1542
PNNL OpenMP	8	63090182	37.5853
PNNL OpenMP	16	63090182	22.0257
PNNL OpenMP	32	63090182	13.1482
PNNL OpenMP	64	63090182	9.055
Gunrock	1xP100 GPU	63090182	**0.060**
GraphBLAS	1xP100 GPU	63090182	0.366

5M edge subgraph (|U|=34395 |V|=20402 |E|=5M)

implementation	num_threads	nnz_out	elapsed_seconds
scipy	1	157071858	10.1052
PNNL OpenMP	1	157051456	357.511
PNNL OpenMP	2	157051456	309.723
PNNL OpenMP	4	157051456	218.519
PNNL OpenMP	8	157051456	113.987

implementation	num_threads	nnz_out	elapsed_seconds
PNNL OpenMP	16	157051456	57.4606
PNNL OpenMP	32	157051456	38.1186
PNNL OpenMP	64	157051456	29.0056
Gunrock	1xP100 GPU	157051456	**0.3349**
GraphBLAS	1xP100 GPU	157051456	1.221

MovieLens-20M graph (|U|=138493 |V|=26744 |E|=20M)

implementation	num_threads	nnz_out	elapsed_seconds
scipy	1	286857534	39.181
PNNL OpenMP	1	286830790	*I killed before finish*
PNNL OpenMP	2	286830790	1109.32
PNNL OpenMP	4	286830790	727.224
PNNL OpenMP	8	286830790	358.708
PNNL OpenMP	16	286830790	188.701
PNNL OpenMP	32	286830790	102.964
PNNL OpenMP	64	286830790	163.731
Gunrock	1xP100 GPU	286830790	*out-of-memory*
GraphBLAS	1xP100 GPU	286830790	**5.012**

Takeaway: When the graph is small enough, Gunrock graph projection is fastest, followed by GraphBLAS (approx. 5x slower). The PNNL OpenMP implementation is consistently substantially slower than the single threaded scipy implementation, even when using 32+ threads.

RMAT

Next we test on a scale 18 RMAT graph. This is *not* a bipartite graph, but the graph projection algorithm can still be applied.

This graph was chosen because it was used in benchmarks in PNNL's gitlab repo. However, their command line parameters appear to be incorrect, so our results here are substantially different than reported in their README.

RMAT-18 (|V|=174147 |E|=7600696)

implementation	num_threads	nnz_out	elapsed_seconds
scipy	1	2973926895	150.869
PNNL OpenMP	1	2973752748	*I killed before finish*
PNNL OpenMP	2	2973752748	*I killed before finish*
PNNL OpenMP	4	2973752748	812.453
PNNL OpenMP	8	2973752748	677.582

implementation	num_threads	nnz_out	elapsed_seconds
PNNL OpenMP	16	2973752748	419.468
PNNL OpenMP	32	2973752748	369.278
PNNL OpenMP	64	2973752748	602.69
Gunrock	1xP100 GPU	2973752748	*out-of-memory*
GraphBLAS	1xP100 GPU	2973752748	**26.478**

Takeaway: GraphBLAS is approx. 5.7x faster than scipy, the next fastest implementation. Again, the PNNL OpenMP implementation is substantially faster than the single-threaded scipy implementation.

When the dataset can fit into memory, Gunrock is ~ 4x faster than GraphBLAS. Since the two implementations use slightly different algorithms, it's hard to tell where the Gunrock speedup comes from. Our hunch is that Gunrock's superior load balancing gives better performance than GraphBLAS, but this is an interesting topic for further research.

Performance information

Gunrock

- Results of profiling indicate that the Gunrock implementation is bound by memory latency.
- The device memory bandwidth is 297 GB/s – within the expected range for Gunrock graph analytics.
- 92% of the runtime is spent in the advance operator (pseudocode in implementation summary)

GraphBLAS

- 99% of time is spent in cuSPARSE's `csrgemm` routines

Next Steps

Alternate approaches

As mentioned above, it would be worthwhile to implement a Gunrock version that does not require allocating the `|V|x|V|` array. It should be possible to achieve this by implementing the same kind of two-pass approach that cuSPARSE uses for `spgemm` – one pass computes the CSR offsets, then column and data values are inserted at the appropriate locations.

Gunrock implications

Gunrock does not natively support bipartite graphs, but it is straightforward for programmers to implement algorithms that expect bipartite graphs by

keeping track of the number of nodes in each node set. However, for multi-GPU implementations, the fact that a graph is bipartite may be useful for determining the optimal graph partitioning.

Notes on multi-GPU parallelization

Multiple GPU support for GraphBLAS is on the roadmap. Unlike the Seeded Graph Matching problem – which requires the `mxm`, `mxv` and `vxm` primitives, which in turn necessitates possible changes in data layout – this problem only requires the `mxm` primitive, so multiple GPU support for graph projections is easier than for SGM.

Even though extending matrix multiplication to multiple GPUs can be straightforward, doing so in a backend-agnostic fashion that abstracts away the placement (i.e. which part of matrix A goes on which GPU) may still be quite challenging.

Further discussion can be found here.

Notes on dynamic graphs

This workflow does not have an explicit dynamic component. The graph projection operation seems like it would be fairly straightforward to adapt to dynamic graphs – as nodes/edges are added to `G`, we create/increment the weight of the appropriate edges in `H`. However, this adds an additional layer of complexity to the memory allocation step, as we can't use the two-pass approach to allocate memory conservatively.

Notes on larger datasets

If the dataset were too big to fit into the aggregate GPU memory of multiple GPUs on a node, then two directions can be taken in order to be able to tackle these larger datasets:

- Out-of-memory: Compute using part of the dataset at a time on the GPU, and save the completed result to CPU memory. This method is slower than distributed, but cheaper and easier to implement.
- Distributed memory: If GPU memory of a single node is not enough, use multiple nodes. This method can be made to scale for infinitely large datasets provided the implementation is good enough (faster than out-of-memory, but more expensive and difficult).

Notes on other pieces of this workload

This workload does not involve any non-graph components.

Chapter 7

Seeded Graph Matching (SGM)

From Fishkind et al.:

> Given two graphs, the graph matching problem is to align the two vertex sets so as to minimize the number of adjacency disagreements between the two graphs. The seeded graph matching problem is the graph matching problem when we are first given a partial alignment that we are tasked with completing.

That is, given two graphs `A` and `B`, we seek to find the permutation matrix `P` that maximizes the number of adjacency agreements between `A` and `P * B * P.T`, where `*` represents matrix multiplication. The algorithm Fishkind et al. propose first relaxes the hard 0-1 constraints on `P` to the set of doubly stochastic matrices (each row and column sums to 1), then uses the Frank-Wolfe algorithm to minimize the objective function `sum((A - P * B * P.T) ** 2)`. Finally, the relaxed solution is projected back onto the set of permutation matrices to yield a feasible solution.

Summary of Results

SGM is a fruitful workflow to optimize, because the existing implementations were not written with performance in mind. By making minor modifications to the algorithm that allow use of sparse data structures, we enable scaling to larger datasets than previously possible.

The application involves solving a linear assignment problem (LSAP) as a subproblem. Solving these problems on the GPU is an active area of research – though papers have been written describing high-performance parallel LSAP solvers, reference implementations are not available. We implement a GPU LSAP solver via Bertsekas' auction algorithm, and make it available as as standalone library.

SGM is an approximate algorithm that minimizes graph adjacency disagreements via the Frank-Wolfe algorithm. Certain uses of the auction algorithm can introduce additional approximation in the gradients of the Frank-Wolfe iterations. An interesting direction for future work would be a rigorous study of the effects of this kind of approximation on a variety of different graph tolopogies. Understanding of those dynamics could allow further scaling beyond what our current implementations can handle.

Summary of Gunrock Implementation

The SGM algorithm consists of:

- several matrix-matrix and matrix-vector multiplies
- solving a linear assignment problem at each iteration
- computing the trace of matrix products (e.g., `trace(A * B)`)

The formulation of SGM in the HIVE reference implementation does not take advantage of the sparsity of the problem. This is due to two algorithmic design choices:

1. In order to penalize adjacency disagreements, they convert the 0s in the input matrices `A` and `B` to -1s
2. They initialize the solution matrix `P` near the barycenter of the Birkhoff polytope of doubly-stochastic matrices, so almost all entries are nonzero.

In order to take advantage of sparsity and make SGM more suitable for HIVE analysis, we make two relatively small modifications to the SGM algorithm:

1. Rework the equations so we can express the objective function as a function of the sparse adjacency matrices plus some simple functions of node degree
2. Initialize `P` to a vertex of the Birkhoff polytope

A nice property of the Frank-Wolfe algorithm is that the number of nonzero entries in the intermediate solutions grows slowly – after the `n`th step, the solution is the convex combination of (at most) `n` vertices of the polytope (e.g., permutation matrices). Thus, starting from a sparse initialization point means that all of the Frank-Wolfe steps are fairly sparse.

The reference implementation uses the Jonker-Volgenant algorithm to solve the linear assignment subproblem. However, the JV algorithm (and the similar Hungarian algorithm) do not admit straightforward parallel implementations. Thus, we replace the JV algorithm with Bertsekas' auction algorithm, which is much more natural to parallelize.

Because SGM consists of linear algebra plus an LSAP solver, we implement it outside of the Gunrock framework, using a GPU GraphBLAS implementation from John Owens' lab, as well as the CUDA CUB library.

How To Run This Application on DARPA's DGX-1

Prereqs/input

```
git clone --recursive https://github.com/owensgroup/csgm.git
cd csgm

# build
make clean
make

# make data (A = random sparse matrix, B = permuted version of A, except first `num-seeds`
python data/make-random.py --num-seeds 100
wc -l data/{A,B}.mtx
```

Running the application

Command:

```
./csgm --A data/A.mtx --B data/B.mtx --num-seeds 100 --sgm-debug 1
```

Output:

```
===== iter=0 ================================
APB_num_entries=659238
counter=17
run_num=0 | h_numAssign=4096 | milliseconds=21.3737
APPB_trace = 196
APTB_trace = 7760
ATPB_trace = 7760
ATTB_trace = 109460
ps_grad_P  = 393620
ps_grad_T  = 16124208
ps_gradt_P = 407896
ps_gradt_T = 15879704
alpha      = -29.4213314
falpha     = 224003776
f1         = -15486084
num_diff   = 448756
------------
f1 < 0
iter_timer=74.0615005
...
===== iter=2 ================================
APB_num_entries=13464050
```

```
counter=1
run_num=0 | h_numAssign=4096 | milliseconds=5.71267223
APPB_trace = 333838
APTB_trace = 333838
ATPB_trace = 333838
ATTB_trace = 333838
ps_grad_P  = 16777216
ps_grad_T  = 16777216
ps_gradt_P = 16777216
ps_gradt_T = 16777216
alpha      = 0
falpha     = -1
f1         = 0
num_diff   = 0
------------
iter_timer=45.222271
total_timer=170.153473 | num_diff=0
```

Note: Here, the final `num_diff=0` indicates that the algorithm has found a perfect match between the input graphs.

Output

When run with `--sgm-debug 1`, we output information about the quality of the match in each iteration. The most important number is `num_diff`, which gives the number of disagreements between `A` and `P * B * P.T`. `num_diff=0` indicates that SGM has found a perfect matching between `A` and `B` (eg, there are no adjacency disagreements).

This implementation is validated against the HIVE reference implementation. Additionally, since the original reference implementation code was posted, Ben Johnson has worked with Johns Hopkins to produce other more performant implementations of SGM, found here.

Performance and Analysis

Given:

- two input graphs `A` and `B`
- a set of seeds `S`
- some parameters controlling convergence (`num_iters`, `tolerance`)

Performance is measured by runtime of the entire SGM procedure as well as the final number of adjacency disagreements between `A` and `P * B * P.T`.

Per-iteration runtime is not necessarily meaningful, because different iterations present dramatically more difficult LSAP instances than others. In particular, the LSAP solver in the first iteration tends to take 10-100x longer than in subsequent iterations.

Bertsekas' auction algorithm allows us to make a tradeoff between runtime and accuracy. With appropriate parameter settings, it produces the exact same answer as the JV or Hungarian algorithms. However, with different parameter settings, the auction algorithm may run substantially faster (>10x), at the cost of a lower quality assignment. Since SGM is already an approximate algorithm, *we do not currently know the SGM's sensitivity to this kind of approximation.* Experiments to explore these tradeoffs would be an interesting direction for future research. In general, we run our SGM implementation with some approximation, and thus we rarely produce the exactly optimal solution for the LSAP subproblems. However, we often produce *SGM solutions* of comparable quality to those SGM implementations that exactly solve the LSAP subproblems.

Implementation limitations

SGM is only appropriate for pairs of graphs w/ some kind of correspondence between the nodes. This could be an explicit correspondance (users on Twitter and users on Instagram, people in a cell phone network and people on an email network), or an implicit correspondence (two people play similar roles at similar companies).

Currently, our implementation of SGM only supports undirected graphs – an extension to directed graphs is mathematically straightforward, but requires a bit of code refactoring. We have only tested on unweighted graphs, though the code should also work on weighted graphs out-of-the-box.

We also currently assume that the number of nodes in `A` and `B` are identical. Fishkind et al. discuss various padding schemes to address the cases where `A` and `B` have a different number of nodes, but we assume all of these could be done in a preprocessing step before the graph is passed to our SGM implementation.

At the moment, the primary scaling bottleneck is that we allocate two `|V|x|V|` arrays as part of the LSAP auction algorithm. These could be replaced w/ 3 `|E|` arrays without much effort.

Currently, the auction algorithm does not take advantage of all available parallelism. Each CUDA thread gets a row of the cost matrix, and then does a serial reduction across the entries of the row. As the auction algorithm runs, the number of "active" rows rapidly decreases. This means that the majority of auction iterations have a small number of active rows, and thus use a small number of GPU threads. We could better utilize the GPU by using multiple threads per row. We have a preliminary implementation of this using the CUB library, but it has not been tested on various relevant edge cases.

Preliminary experiments suggest the CUB implementation may be 2–5x faster than our current implementation.

Comparison against existing implementations

Python SGM code

The original SGM implementation is a single-threaded R program. Preliminary tests on small graphs show that this implementation is not very performant. As part of other work, we've written a modular SGM package that allows the programmer to plug in different backends for the LSAP solver and the linear algebra operations. This package includes modes that transform the SGM problem to take advantage of sparsity to improve runtime. In particular, we benchmark our CUDA SGM implementation against the `scipy.sparse.jv` mode, which uses `scipy` for linear algebra and the gatagat/lap implementation of the JV algorithm for the LSAP solver.

Experiments

Connectome

The connectome graphs are generated from brain MRIs. Nodes represent a voxel in the MRI and edges indicate some kind of flow between voxels. By using voxels of different spatial resolutions, the researchers that collected the data produced pairs of graphs at a variety of sizes (smaller voxel = larger graph). Each graph in a pair represents one hemisphere of the brain. Thus, we know there is an actual approximate correspondence between nodes.

Note that these graphs are already partially aligned – the distance between the input graphs is far smaller than would be expected by chance. We attempt to use SGM to further improve this initial alignment, using all nodes as "soft seeds" (see Fishkind et al. for further discussion).

The size of the connectome graphs we consider are as follows:

name	num_nodes	num_edges
DS00833	00833	12497
DS01216	01216	19692
DS01876	01876	34780
DS03231	03231	64662
DS06481	06481	150012
DS16784	16784	445821
DS72784	72784	2418304

In the tables below, `orig_dist` indicates the number of adjacency disagreements between `A` and `B`, and `final_dist` indicates the number of adjacency disagreements between `A` and `P * B * P.T` w/ `P` found by SGM. We run CSGM

w/ two values of `eps`, which controls the precision of the auction algorithm (lower values = more precise but slower).

Python runtimes

name	orig_dist	final_dist	time_ms
DS00833	11650	11486	122.656
DS01216	20004	19264	278.739
DS01876	38228	36740	2275.141
DS03231	78058	73282	8900.371
DS06481	201084	183908	97658.378
DS16784	677754	593590	920436.387

CSGM runtimes

eps	name	orig_dist	final_dist	time_ms	speedup
1.0	DS00833	11650	11538	181.481	0.6
1.0	DS01216	20004	19360	324.908	0.8
1.0	DS01876	38228	36936	807.148	2.8
1.0	DS03231	78058	73746	3078.78	2.9
1.0	DS06481	201084	184832	9056.55	10.7
1.0	DS16784	677754	596370	42220.5	21.8
1.0	DS72784	—	—	OOM	—
0.5	DS00833	11650	11466	378.056	0.3 *
0.5	DS01216	20004	19288	965.915	0.3
0.5	DS01876	38228	36764	1258.65	1.8
0.5	DS03231	78058	73346	6257.87	1.4
0.5	DS06481	201084	183796	25931.2	3.7 *
0.5	DS16784	677754	592822	120799	7.6 *
0.5	DS72784	—	—	OOM	—

Takeaway: For small graphs (`|U| < ~2000`) the Python implementation is faster. However, as the size of the graph grows, CSGM becomes significantly faster – up to 20x faster in the low accuracy setting and up to 7.6x faster in the higher accuracy setting. Also, though in general the auction algorithm does not compute exact solutions to the LSAP, in several cases CSGM's accuracy outperforms the Python implementation, which uses an exact LSAP solver – these are denoted with a `*`.

Kasios

The Kasios call graph shows the communication pattern inside of a corporation – nodes represent a person and edges indicate that two people spoke on the phone.

The whole graph has 10M edges and 500K nodes, which is too large for any of our SGM implementations to handle. Thus, we sample subgraphs of size 2**10 - 2**15 by running a random walk until the desired number of nodes are observed, and then extracting the subgraph induced by those nodes. We generate pairs of graphs by random permutations (except for the first `num_seeds=100` nodes). Interestingly, with 100 seeds, SGM can almost always perfectly "reidentify" the permuted graph.

Python runtimes

num_nodes	orig_dist	final_dist	time_ms
1024	66636	0	242
2048	208144	0	1275
4096	580988	0	6530
8192	1449712	0	30763
16384	3235356	0	118072
32768	6587656	0	479181

CSGM runtimes (`eps=1`)

num_nodes	orig_dist	final_dist	time_ms	speedup
1024	66636	0	182.445	1.3
2048	208144	4	310.566	4.1
4096	580988	4	1026.07	6.4
8192	1449712	8	3108.9	9.9
16384	3235356	16	4926.85	24.0
32768	6587656	OOM!	—	—

Takeaway: Similar to in the connectome experiments, we see the advantage of CSGM increase as the graphs grow larger. In these examples, CSGM does not *quite* match the performance of the exact LSAP solver. However, this could be addressed by tuning and/or scheduling the `eps` parameter.

Performance limitations

- Results from profiling indicate that all of the SGM kernels are memory latency bound.
- 50% of time is spent in sparse-sparse matrix-multiply (SpMM)
- 39% of time is spent in the auction algorithm. Of this 39%:
  - 65% of time is spent in `run_bidding`
  - 26% of time is spent in `cudaMemset`
  - 9% of time is spent in `run_assignment`

Next Steps

Alternate approaches

It would be worthwhile to look into parallel versions of the Hungarian or JV algorithms, as even a single-threaded CPU version of those algorithms is somewhat competitive with Bertseka's auction algorithm implemented on the GPU. It's unclear whether it would be better to implement parallel JV/Hungarian as multi-threaded CPU programs or GPU programs. If the former, SGM would be a good example of an application that makes good use of both the CPU (for LSAP) and GPU (for SpMM) at different steps.

Gunrock implications

N/A

Notes on multi-GPU parallelization

GraphBLAS

Multiple GPU support for GraphBLAS is on the roadmap. This will involve dividing the dataset across multiple GPUs, which can be challenging, because the GraphBLAS primitives required by SGM (`mxm`, `mxv` and `vxm`) have optimal layouts that vary depending on data and each other. There will need to be a tri-lemma between inter-GPU communication, layout transformation, and compute time for optimal vs. sub-optimal layout.

Although extending matrix-multiplication to multiple GPUs can be straightforward, doing so in a backend-agnostic fashion that abstracts away the placement (i.e., which part of matrix `A` goes on which GPU) from the user may be quite challenging. This can be done in two ways:

1. Manually analyze the algorithm and specify the layout in a way that is application-specific to SGM (easier, but not as generalizable)
2. Write a sophisticated runtime that will automatically build a directed acyclic graph (DAG), analyze the optimal layouts, communication volume and required layout transformations, and schedule them to different GPUs (difficult and may require additional research, but generalizable)

Auction algorithm

The auction algorithm can be parallelized across GPUs in several ways:

1. Move data onto a single GPU and run the existing auction algorithm (simple, but not scalable).
2. Bulk-synchronous algorithm: Run auction kernel, communicate, then run next iteration of auction kernel (medium difficulty, scalable).

3. Asynchronous algorithm: Run auction kernel and communicate to other GPUs from within kernel (difficult, most scalable).

Notes on dynamic graphs

Real-world applications of SGM to eg. social media or communications networks often involve dynamic graphs. Application of SGM to streaming garphs could be a very interesting new research direction. To our knowledge, this problem has not been studied by the JHU group responsible for the algorithm. Given an initial view of two graphs, we could compute an initial match, and then update the match via a few iterations of SGM as new edges arrive.

Notes on larger datasets

If the dataset were too big to fit into the aggregate GPU memory of multiple GPUs on a node, then two directions can be taken in order to be able to tackle these larger datasets:

1. Out-of-memory: Compute using part of the dataset at a time on the GPU, and save the completed result to CPU memory. This method is slower than distributed, but cheaper and easier to implement.
2. Distributed memory: If the GPU memory on a single node is not enough, use multiple nodes. This method can be made to scale for arbitrarily large datasets provided the implementation is good enough (faster than out-of-memory, but more expensive and difficult).

Notes on other pieces of this workload

There are no non-graph pieces of the SGM workload.

How this work can lead to a paper publication

Ben and Carl think this work can lead to a nice paper, because there aren't a lot of highly optimized parallel Linear Assignment Problem (LAP) solvers. A lot of the research Ben could find from 20+ years ago tends to assume that the input matrices are uniformly random. However, our use case is on cost matrices formed by the dot product of sparse matrices (so the `(i, j)`th entry is the number of neighbors node i and j have in common), which has a totally different distribution (closer to a power law). There may be some optimizations we can find that target this distribution (similar to how direction-optimized BFS targets power law graphs).

There is potential research in tie-breaking for the auction algorithm. In one of the popular Python/C++ LAP solvers, they're clearly not handling ties smartly, and the runtime can be improved ~10x by adding random values in a specific way. For these types of data, we find a lot of people assuming there

aren't many ties. But with graphs, Ben notices many entries are ties, so some randomization is clearly beneficial.

Further development on the standalone auction algorithm will happen here. This will include porting the current implementation to CUDA CUB to take better advantage of available parallelism, as well as writing Python bindings for ease of use.

title: Vertex Nomination (HIVE)

toc_footers: - Gunrock: GPU Graph Analytics - Gunrock © 2018 The Regents of the University of California.

search: true

full_length: true

Chapter 8

Vertex Nomination

The vertex nomination (VN) workflow is an implementation of the kind of algorithm discussed in Coppersmith and Priebe.

Often, we have an attributed graph where we know of some "interesting" nodes, and we want to rank the rest of the nodes by their likelihood of being interesting. Coppersmith and Priebe propose a general framework for using both node attributes ("content") and network features ("context") to rank nodes. The specific content, context, and fusion functions can be arbitrary, user-defined functions.

Summary of Results

The term "vertex nomination" covers a variety of different node ranking schemes that fuse "content" and "context" information. The HIVE reference code implements a "multiple-source shortest path" context scoring function, but uses a very suboptimal algorithm. By using a more efficient algorithm, our serial CPU implementation achieves 1-2 orders of magnitude speedup over the HIVE implementation and our GPU implementation achieves another 1-2 orders of magnitude on top of that. Implementation was straightforward, involving only a small modification to the existing Gunrock SSSP app.

Summary of Gunrock Implementation

Since HIVE is focused on graph analytics, the content scoring function is not relevant, and we only implement the context scoring function. Coppersmith and Priebe propose a number of possible network statistics that can be used for context scoring. The HIVE reference implementation ranks each node `u` in a graph `G = (U, E)` by the minimum distance from `u` to a node in a set of seed nodes `S`. This is the VN variant we have implemented in Gunrock.

This choice of context scoring function ends up being nearly identical to a single source shortest paths (SSSP) problem. The one difference is that we start from the set of seed nodes `S` instead of single node.

Because of this similarity to SSSP, the Gunrock VN implementation consists of making a minor modification to the Gunrock SSSP implementation, so that it can accept a list of source nodes instead of a single source node. Thus, the core of the VN algorithm is a Gunrock advance operator implementing a parallel version of the Bellman-Ford algorithm. Specifically, in `python`:

```
class IterationLoop(BaseIterationLoop):
    def _advance_op(self, src, dest, problem, enactor_stats):
        src_distance = problem.distances[src]
        edge_weight  = problem.edge_weights[(src, dest)]
        new_distance = src_distance + edge_weight

        old_distance = problem.distances[dest]
        problem.distances[dest] = min(old_distance, new_distance)

        return new_distance < old_distance

    def _filter_op(self, src, dest, problem, enactor_stats):
        if problem.labels[dest] == enactor_stats['iteration']:
            return False

        problem.labels[dest] = enactor_stats['iteration']
        return True
```

Note we could have used the Gunrock SSSP implementation directly by

1. adding a dummy node `d` to `G`
2. adding an edge `(d, s)` between `d` and each node `s` in `S` with `weight(d, s) = 0`
3. running SSSP from `d`

How To Run This Application on DARPA's DGX-1

Prereqs/input

```
git clone --recursive https://github.com/gunrock/gunrock -b dev-refactor
cd gunrock/tests/vn
cp ../../gunrock/util/gitsha1.c.in ../../gunrock/util/gitsha1.c
make clean
make
```

Application specific parameters

```
--src
  Comma separated list of seed nodes (eg, `0,1,2`) OR `random` (see below)
--srcs-per-run
  If `src=random`, number of randomly chosen source nodes per run
--num-runs
  Number of runs
```

Example command

```
./bin/test_vn_9.1_x86_64 \
    --graph-type market \
    --graph-file ../../dataset/small/chesapeake.mtx \
    --src random \
    --srcs-per-run 10 \
    --num-runs 2
```

Example output

```
Loading Matrix-market coordinate-formatted graph ...
  Reading meta data from ../../dataset/small/chesapeake.mtx.meta
  Reading edge lists from ../../dataset/small/chesapeake.mtx.coo_edge_pairs
  Assigning 1 to all 170 edges
  Substracting 1 from node Ids...
  Edge doubleing: 170 -> 340 edges
  graph loaded as COO in 0.090935s.
Converting 39 vertices, 340 directed edges ( ordered tuples) to CSR format...Done (0s).
Degree Histogram (39 vertices, 340 edges):
    Degree 0: 0 (0.000000 %)
    Degree 2^0: 0 (0.000000 %)
    Degree 2^1: 1 (2.564103 %)
    Degree 2^2: 22 (56.410256 %)
    Degree 2^3: 13 (33.333333 %)
    Degree 2^4: 2 (5.128205 %)
    Degree 2^5: 1 (2.564103 %)


------------------------
------------------------
Run 0 elapsed: 0.025034 ms, srcs = 21,19,3,28,20,25,23,26,13,38
------------------------
------------------------
Run 1 elapsed: 0.021935 ms, srcs = 21,15,5,23,3,29,22,26,33,4
============================================
 mark-pred=0 advance-mode=LB
Using advance mode LB
```

```
Using filter mode CULL

-------------------------
0  1   0   queue3    oversize :  234 ->  246
0  1   0   queue3    oversize :  234 ->  246
-------------------------
Run 0 elapsed: 0.442028 ms, srcs = 21,19,3,28,20,25,23,26,13,38, #iterations = 3
-------------------------
-------------------------
Run 1 elapsed: 0.329971 ms, srcs = 21,15,5,23,3,29,22,26,33,4, #iterations = 3
Distance Validity: PASS
First 40 distances of the GPU result:
[0:1 1:1 2:1 3:0 4:0 5:0 6:1 7:1 8:2 9:2 10:2 11:2 12:2 13:1 14:1 15:0 16:1 17:1 18:1 19:2
First 40 distances of the reference CPU result.
[0:1 1:1 2:1 3:0 4:0 5:0 6:1 7:1 8:2 9:2 10:2 11:2 12:2 13:1 14:1 15:0 16:1 17:1 18:1 19:2

[vn] finished.
 avg. elapsed: 0.386000 ms
 iterations: 3
 min. elapsed: 0.329971 ms
 max. elapsed: 0.442028 ms
 rate: 0.880830 MiEdges/s
 src: 3
 nodes_visited: 39
 edges_visited: 340
 load time: 113.31 ms
 preprocess time: 974.719000 ms
 postprocess time: 0.562906 ms
 total time: 976.519108 ms
```

Expected Output

Currently, the VN app does not write any output to disk. It prints runtime statistics and the results of a correctness check. A successful run will print `Distance Validity: PASS` in the output.

Validation

The Gunrock VN implementation was tested against the HIVE reference implementation to verify correctness. We also implemented a CPU reference implementation inside of the Gunrock VN app, with results that match the HIVE reference implementation.

Additionally, for ease of exposition, we implemented a pure Python version of the Gunrock algorithm that lets people new to Gunrock see the relevant logic without all of the complexity of C++/CUDA data structures, memory management, etc. In this case, we already knew how to implement VN using

Gunrock primitives. However, in other cases, where we're writing a Gunrock app from scratch, translation from some arbitrary serial implementation to `advance`/`filter`/`compute` can be complex and involve some trial and error to handle edge cases. In those cases, `pygunrock` has proven to be a useful tool for rapid prototyping and debugging.

Our implementation of VN is a deterministic algorithm, so all correct solutions have the same accuracy/quality.

Performance and Analysis

Performance is measured by the runtime of the app, given:

- an input graph `G=(U, E)`
- a set of seed nodes (or size/number of random seed sets)

Other implementations

Python reference implementation

The Python + SNAP reference implementation can be found here. This is a very naive implementation of the context function – rather than running the SSSP variant we describe above, it runs a separate BFS from each node `s` in the seed set `S` to each node `u` in `U`. Thus, its algorithmic complexity is approximately `|S|x|U|` times larger than the Gunrock implementation.

Performer OpenMP implementations

We were unable to locate any C/OpenMP implementation of VN from TA1/TA2 performers at the time of writing. (2018-10-20)

Gunrock CPU implementation

For correctness checking, we implement VN in single-threaded C++ within the Gunrock testing framework. This is a serial implementation of Djikstra's algorithm using a CSR graph representation and `std::priority_queue`. We expect this to be substantially faster than the HIVE Python+SNAP reference implementation due to superior algorithmic complexity.

Experiments

HIVE Enron dataset (|U|=15056, |E|=57075)

The Enron graph is a graph of email communications between employees of Enron.

The HIVE reference implementation implementation does 10 runs w/ 5 random seeds each on the Enron email dataset. Results are as follows:

implementation	elapsed_ms (avg of 10 runs)
Python+SNAP	4115.545
Gunrock CPU	7.305
Gunrock GPU	**0.921**

Takeaway: Due to improved algorithmic efficiency, the Gunrock CPU implementation is approximately 563x faster than the HIVE reference implementation. The Gunrock GPU implementation is approximately 7.9x faster than the Gunrock CPU implementation. However, this dataset may be too small for these numbers to be very precise.

Hollywood-2009 graph (|U|=1139905, |E|=57515616)

The Hollywood-2009 graph is a graph of Hollywood movie actors, where nodes are actors and edges indicate two actors appear in a movie together.

implementation	elapsed_ms (avg of 10 runs)
Python+SNAP	*> 10 minutes*
Gunrock CPU	2035.45
Gunrock GPU	**13.793**

Takeaway: Here, the Gunrock GPU implementation is approximately 150x faster than the Gunrock CPU implementation. The HIVE reference implementation did not finish in 10 minutes.

Indochina-2004 graph (|U|=7414866, |E|= 191606827)

The Indochina-2004 graph is an internet hyperlink graph, generated by a crawl of Asian country domains.

implementation	elapsed_ms (avg of 10 runs)
Python+SNAP	*> 10 minutes*
Gunrock CPU	9079.216
Gunrock GPU	**22.743**

Takeaway: Here, the Gunrock GPU implementation is approximately 400x faster than the Gunrock CPU implementation. The HIVE reference implementation did not finish in 10 minutes.

Implementation limitations

The size of the graph that can be processed will (usually) be limited by the number of edges `|E|` in the input graph. The VN algorithm only allocates an additional 1–3 arrays of size `|U|`, and thus does not require a large amount of storage for temporary data.

The Gunrock VN algorithm works on weighted/unweighted directed/undirected graphs. No particular graph topology or node/edge metadata is required. In general, VN would be run on graphs with node and/or edge attributes, but since our Gunrock app only implements context scoring, we are not subject to those restrictions.

Performance limitations

- Like SSSP, VN is bound by device memory latency.
- Profiling indicates that 64% of time is spent in the Gunrock `advance` operator and 20% of time is spent in in the `filter` operator (pseudocode above).
- The device memory bandwidth is 271 GB/s – within the expected range for Gunrock graph analytics. Random memory access means that we don't expect to get close to the reported maximum memory bandwidth.

Next Steps

Alternate approaches/further work

Because of its similarity to SSSP, this implementation of VN is fairly hardened. However, given more time, we could implement more variations on context similiarity as described in Coppersmith and Priebe's paper. Given the range of potential context similarity functions, this could involve implementing a wide variety of Gunrock operators.

Gunrock implications

This was a straightforward adapatation of an existing Gunrock app. SSSP is also one of the simpler apps – only one advance/filter operation without complex logic – so implementing VN was not very difficult. All of the core logic in VN is identical to SSSP.

Notes on multi-GPU parallelization

Discussion of multi-GPU scalability of VN can be found here.

Notes on dynamic graphs

The reference implementation does not cover a dynamic graph version of this workflow, though one could imagine having a static set of seed nodes and a

streaming graph on which one would like to compute context scores in real time.

Notes on larger datasets

If the datasets are larger than a single or multi-GPU's aggregate memory, the straightforward solution would be to let Unified Virtual Memory (UVM) in CUDA automatically handle memory movement. Declaring the entire graph as managed memory for a single GPU implementation, will allow the users to simply oversubscribe for 1 GPU, and queue vertices and edges in from the host memory as needed (this will be very slow). Further optimizations can be done, where instead of utilizing host memory, we can leverage a multi-GPU implementation and move the entire graph over to device memory, using NVLink to move data between the devices' global memory. This can be further optimized by using MemAdvise hints such as pinning the memory to GPU's local memory where it is likely to be used, or create a direct map to all other GPUs to avoid page faulting on first touch.

Notes on other pieces of this workload

The context scoring component of vertex nomination is incredibly general, and could include versions ranging in complexity from simple Euclidean distance metrics to the output of complex deep learning pipelines. If we were to integrate these kinds of components more closely w/ Gunrock, we'd likely need to use other CUDA libraries (cuBLAS, cuDNN, etc.) as well as interface w/ higher-level machine learning libraries (TensorFlow, PyTorch, etc.).