Sandbox

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

UC Davis

Contents

Chapter 1

Scaling analysis for HIVE applications

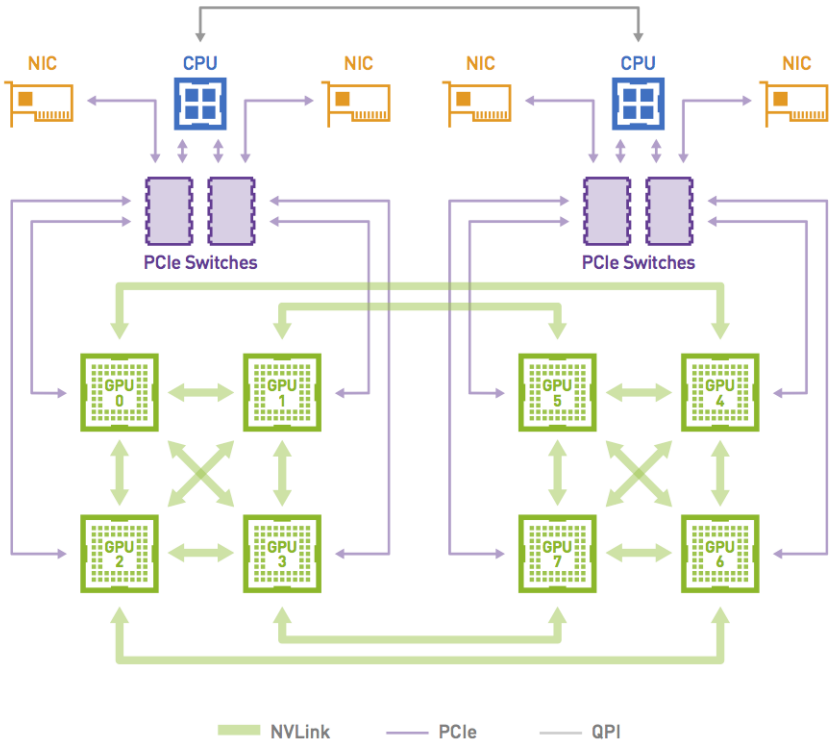
The purpose of this study is to understand how the HIVE v0 applications would scale out on multiple GPUs, with a focus on the DGX-1 platform. Before diving into per-application analysis, we give a brief summary of the potential hardware platforms, the communication methods and models, and graph partitioning schemes.

Hardware platforms

There are at least three kinds of potential multi-GPU platforms for HIVE apps. They have different GPU communication properties, and that may affect the choice of communication models and/or graph partitioning schemes, thus may have different scaling results.

DGX-1

The DGX-1 with P100 GPUs has 4 NVLink lanes per GPU, connected as follows.



Each of the NVLink links runs at 20 GBps per direction, higher than PCIe 3.0 x16 (16 GBps for the whole GPU). But the topology is not all-to-all, and GPUs may not be able to access every other GPU's memory. For example, GPU0 can't use peer access on GPUs 5, 6 and 7. This makes implementations using peer access more complex than a full all-to-all topology. DGX-1 with V100 GPUs increases the NVLink speed to 25 GBps per direction per lane, and increases the number of lanes per GPU to 6, but peer accessibility has not been changed. This issue is finally addressed in DGX-2 with the NVSwitch.

Using a benchmark program to test throughput and latency shows the following results. `Self` indicates local GPU accesses, `peer` indicates peer accesses, `host` indicates accesses to the CPU memory via UVM, and `all` indicates accesses to all peer-accessible GPUs. The `regular` operations access the memory in continous places by neighboring threads; in CUDA terms, these operations are coalesced memory accesses. The `random` operations access the memory space randomly, and neighboring threads may be touching memory that are far away from each other; in CUDA terms, these operations are non-coalesced memory accesses. The memory access patterns of graph workflows are a mix of both,

and one main target of kernel optimizations is to make the memory accesses as coalesced as possible. But at the end, depending on the algorithm, some random accesses may be unavoiable. Random accesses across GPUs are particularly slow.

Throughput in GBps:

Operation	Self	Peer	Host	All
Regular read	448.59	14.01	444.74	12.17
Regular write	442.98	16.21	16.18	12.17
Regular update	248.80	11.71	0.0028	6.00
Random read	6.78	1.43	2.39	4.04
Random write	6.63	1.14	3.47E-5	3.82
Random update	3.44	0.83	1.92E-5	2.08

Latency in microseconds (us):

Operation	Self	Peer	Host	All
Regular read	2.12	1.18	1.30	1.49
Regular write	1.74	1.00	13.83	1.01
Regular update	2.43	1.20	79.29	1.44
Random read	3.11	1.08	13.61	1.40
Random write	3.28	1.05	15.88	1.39
Random update	5.69	1.28	21.76	1.38

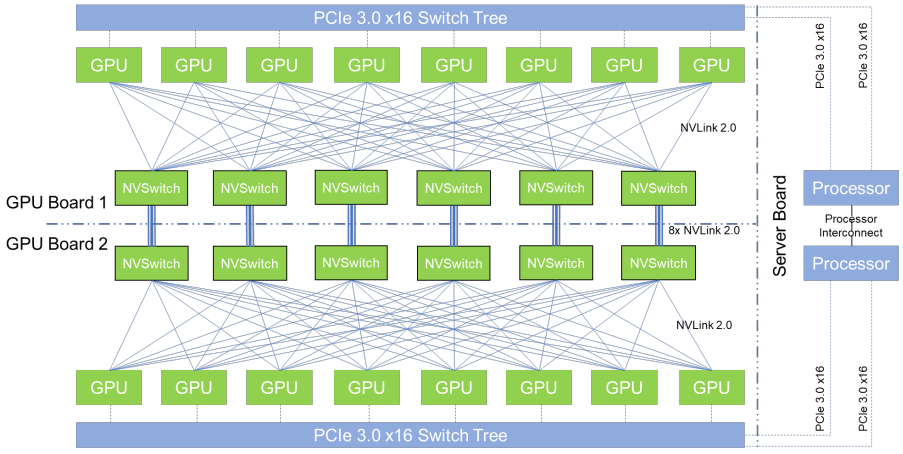
All the regular throughputs are at least 80% of their theoretical upper bounds. The latencies when accessing local GPUs seem odd, but other latencies look reasonable. It's clear that local regular accesses have much higher throughput than inter-GPU connections, about 20 to 30 times from this experiment. The ratio of random accesses are lower, but still at about 5 times. The implication on the scalabilities of graph applications is that for scalable behavior, the local-memory-access-to-communication ratio needs to be at least 10 to 1. Because most graph implementations are memory bound, the computation cost is counted by the number of elements accessed by the kernels; this means the computation to communication ratio should be at least 2.5 operations to 1 byte to exhibit scalable behavior.

Unified virtual memory (UVM) doesn't work as expected for most cases, instead only when the accesses are read only and the data can be duplicated on each GPU.

Otherwise the throughputs are significantly lower, caused by memory migration and going over the PCIe interfaces. The less than 0.5 GBps throughputs from write and update operations via UVM could possibily caused by problems in the UVM support or the way how UVM is configured in the benchmark code. It must be noted that, at the time of testing, the DGX-1 has CUDA 9.1 and NVIDIA driver 390.30 installed, which are more than one year old. A newer CUDA version and NVIDIA driver could potentially improve the UVM performance. Testing using V100 GPUs with CUDA 10.0 and NVIDIA driver 410 shows considerablly better throughputs.

DGX-2

The DGX-2 system has a very different NVLink topology: the GPUs are connected by NVSwitches, and all to all peer accesses are available.



At the time of this report, the DGX-2 is hardly available, and not to us. What we have locally at UC Davis are two Quadro GV100 GPUs directly connected by 4 NVLink2 lanes. Although the setup is much smaller than the DGX-2, it still can provide some ideas on how the inter-GPU communication would perform on the DGX-2.

Throughput in GBps

Operation	Self	Peer	Host	All
Regular read	669.68	76.74	679.52	76.72
Regular write	590.01	85.00	170.113	76.28
Regular update	397.26	39.67	80.00	37.86
Random read	17.39	7.46	17.27	10.51
Random write	13.25	7.96	1.23	7.24
Random update	6.83	3.88	0.68	3.85

Latency in microsecond (us)

Operation	Self	Peer	Host	All
Regular read	0.37	0.47	0.37	0.41
Regular write	0.08	0.08	0.08	0.15
Regular update	0.37	0.47	0.43	0.41
Random read	0.11	0.10	0.10	0.16
Random write	0.13	0.08	0.08	0.15
Random update	0.15	0.09	0.09	0.17

On this machine configuration, the local-to-peer memory throughput ratios, about 8 for regular accesses and about 2 for random accesses, are much lower than the DGX-1. The decreases are mainly from using all the 4 lanes for communication, instead of 1 in the DGX-1 cases. If using only a single lane, the ratios would become 32 and 8, even higher than the DGX-1. The actual effect from the NVSwitch is still unclear, but DGX-2 is expected to have similar scalabilities as DGX-1 for graph applications.

GPU clusters

Going from multiple GPUs within the same node to multiple GPUs across different nodes significantly decreases the inter-GPU throughput. While NVLink runs at 80 GBps per direction per GPU for the DGX-1, the aggregated InfiniBand bandwidth is only 400 Gbps, which is only one twelfth of the aggregated inter-GPU bandwidth. This means the local access-to-communication-bandwidth ratio drops an order of magnitude, making scaling graph applications across nodes a corresponding order of magnitude harder. Using the same approximation method as the DGX-1, a graph implementation needs to have 30 operations / local memory operations for each byte going across the nodes. The implementation focus may need to switch to communication, rather than local computation, to achieve a scalable implementation on GPU clusters.

Communication methods / models

There are multiple ways to move data between GPUs: explicit movement, peer accesses, and unified virtual memory (UVM). They have different performance and implications in implementing graph workflows.

Explicit data movement

The most traditional way to communicate is to explicitly move the data: use `cudaMemcpyAsync` to copy a block of memory from one GPU to another. The source and destination GPUs are not required to be peer accessible. CUDA will automatically select the best route: if GPUs are peer-accessible, the traffic will go through the inter-GPU connection, be it NVLink or PCIe; if they are not peer-accessible, the data will be first copied to CPU memory, and then copied to the destination GPU.

One of the advantage of explicit data movement is throughput. `cudaMemcpyAsync` is highly optimized, and the data for communication are always dense. The throughput should be close to the hardware limit, if the size of data is not too small, say at least a few tens of MB. Explicit data movement also isolates local computation and communication. Because there is no need to consider the data layout or different access latencies from local or remote data, the implementation and optimization of computation kernels can be much simpler. It also enables connection to other communication libraries, such as MPI.

However, explicit memory copy requires that data for communication are packed in a continuous memory space. For most applications, this means additional computation to prepare the data. Since the computing power of GPUs is huge, and graph applications are mostly memory-bound, this extra computation should only have minimal impact on the running time.

Many graph algorithms are written using the bulk synchronous parallel (BSP) model: computations are carried out in iterations, and the results of computation are only guaranteed to be visible after the iteration boundary. The BSP model provides a natural communication point: at the iteration boundary. The current Gunrock multi-GPU framework follows the BSP model.

Depending on the algorithm, there are several communication models that can be used:

- *Peer to host GPU* This communication model is used when data of a vertex or edge on peer GPUs need to be accumulated / aggregated onto the host GPU of the vertex. When the vertex or edge is only adjacent to a few GPUs, it may be beneficial to use direct p2p communication; when the vertex is adjacent to most GPUs, a `Reduce` from all GPUs to the host GPU may be better.
- *Host GPU to peers* This is the opposite to the peer-to-host model. It propagates data of a vertex or edge from its host GPU to all GPUs adjacent to the vertex. Similarly, if the number of adjacent GPUs are small, point-to-point communication should do the work; otherwise, `broadcast` from the host GPU may be better.
- *All reduce* When updates on the same vertex or edge come from many GPUs, and the results are needed on many GPUs, `AllReduce` may be the

best choice. It can be viewed as an `peers to host` followed by an `host to peers` communication. It can also be used without partitioning the graph: it works without knowing or assigning a host GPU to an vertex or edge.

- *Mix all reduce with peers to host GPU or host GPU to peers* This is a mix of `AllReduce` and peer-to-peer communications: for vertices or edges that touch a large number of GPUs, `AllReduce` is used; for other vertices or edges, direct peer-to-peer communications are used. This communication model is coupled with the high / low degree partitioning scheme. The paper "Scalable Breadth-First Search on a GPU Cluster" https://arxiv.org/abs/1803.03922 has more details on this model.

Peer accesses

If a pair of GPUs is peer-accessible, they can directly dereference each other's memory pointers within CUDA kernels. This provides a more asynchronous way for inter-GPU communication: there is no need to wait for the iteration boundary. Atomic operations are also supported if the GPUs are connected by NVLink. The implementation can also be kept simple, since no data preparing and explicit movement is needed. The throughput is also acceptable, although not as good as explicit movement.

However, this essentially forces the kernel to work on a NUMA (non-uniform memory access) domain formed by local and remote GPU memory. Some kernels optimized under the assumption of a flat memory domain may not work well. Peer accesses also give up the opportunity to aggregate local updates on the same vertex or edge before communication, so the actual communication volume may be larger.

Unified virtual memory (UVM)

UVM is similar to peer accesses, as it also enables multiple GPUs to access the same memory space. However, the target memory space is allocated in the CPU memory. When needed on a GPU, a memory page will be moved to the GPU's memory via the page fault mechanism. Updates to the data are cached in GPU memory first, and will eventually be written to the CPU memory. UVM provides a transparent view of the CPU memory on GPU, and significantly reduces the coding complexity if running time is not a concern. It also enables the GPU to process datasets that can't fit in combined GPU memory without explicitly streaming the data from CPU to GPU.

But there are some caveats. The actual placement of a memory page/piece of data relies on hints given by `cudaMemAdvise`, otherwise it will need to be fetched from CPU to GPU when first used by any GPU, and bounces between GPUs when updated by multiple GPUs. The hints essentially come from partitioning the data, but sometimes there is no good way to partition, and data bouncing

is unavoidable. In the worst cases, the data can be moved back to CPU, and any further access will need to go through the slow PCIe bridge again. The performance of UVM is not as good as explicit data movement or peer accesses; in the worst cases, it can be a few orders of magnitude slower. When data is larger than the GPU memory, UVM's throughput drops significantly; as a result, it's easier to code, but it will be slower than streaming the graph for datasets larger than GPU memory.

Graph partitioning scheme

Graphs may be cut into smaller pieces to put them onto multiple GPUs. Duplicating the graph on each may work for some problems, provided the graph is not too large; but duplication is not suitable for all problems, and does not scale in graph size. Graph partitioning is a long-lasting research topic, and we decided to use existing partitioners, such as Metis, instead of implementing some complicated ones of our own. A large number of graphs, especially those with high connectivities, are really difficult to partition; our results suggest that random partitioning works quite well in terms of time taken by the partitioner, load balancing, programming simplicity and still manageable communication cost. By default, Gunrock uses the random partitioner.

What makes a bigger difference is the partitioning scheme: whether the graph is partitioned in 1 dimension, 2 dimensions, or differently for low and high degree vertices.

1D partitioning

1D partitioning distributes the edges in a graph either by the source vertices or the destination vertices. It's simple to work with, and scales well when the number of GPUs are small. It should still work on the DGX-1 with 8 GPUs for a large number of graph applications. But that may approach 1D partitioning's scalability limit.

2D partitioning

2D partitioning distributes the edges by both the source and the destination vertices. When a graph is visualized in a dense matrix representation, 2D partitioning is like cutting the matrix by a 2D grid. 8 GPUs in the DGX-1 may be too small for 2D partitioning to be useful; it may worth trying out on DGX-2 with 16 GPUs. 2D partitioning of sparse graphs may create significant load imbalance between partitions.

High/low degree partitioning

The main idea of high/low degree partitioning is simple: for vertices with low out-degrees and their outgoing edges, distribute edges based on their source

vertices; for vertices with high out-degrees and their outgoing edges, distribute edges based on the destination vertices; if both vertices have high out-degrees, distribute the edge based on the one with lower degree. The result is that low degree vertices are only adjacent to very few GPUs, while high degree vertices' edges are scattered among all GPUs. Graph applications scale very well when using the p2p communication model for low degree vertices, and the `AllReduce` model for high degree vertices.

Scaling of the HIVE applications

The target platform DGX-1 has 8 GPUs. Although that is more than we normally see for single scaling studies, the simple 1D graph partitioning scheme should (in general) still work. The marginal performance improvement by using more GPUs may be insignificant at this scale, and a small number of applications might even see performance decreases with some datasets. However, the 1D partitioning makes the scaling analysis simple and easily understandable. If other partitioning schemes could work better for a specific application, it would be noted.

How scaling is considered

The bandwidth of NVLink is much faster than PCIe 3.0: 20x3 and 25x6 GBps bidirectionally for each Tesla P100 and V100 GPUs respectively in an DGX-1. But compared to the several TFLOPs computing power and several hundred GBps device bandwidth, the inter-GPU bandwidth is still considerably less. For any application to have good scalability, the communication time should be able to be either: 1) hidden by overlap with computation, or 2) kept as a small portion of the computation. The computation-to-communication ratio is a good indicator to predict the scalability: a higher ratio means better scalability. Specific to the DGX-1 system with P100 GPUs, a ratio larger than about 10 to 1 is expected for an app to have at least marginal scalability. For GPUs newer than P100, the computation power and memory bandwidth improve faster than interconnect bandwidth, so the compute-to-communicate ratio needs to grow even more to start seeing positive scaling.

Community Detection (Louvain)

The main and most time-consuming part of the Louvain algorithm is the modularity optimization iterations. The aggregated edge weights between vertices and their respective adjacent communities, as well as the outgoing edge weights of each community, are needed for the modularity optimization. The vertex-community weights can be computed locally, if the community assignments of all neighbors of local vertices are available; to achieve this, a vertex needs to broadcast its new community when the assignment changes. The per-community outgoing edge weights can be computed by `AllReduce`

across all GPUs. The modularity optimization with inter-GPU communication can be done as:

```
Do
    Local modularity optimization;
    Broadcast updated community assignments of local vertices;
    local_weights_community2any := sums(local edges from an community);
    weights_community2any := AllReduce(local_weights_community2any, sum);
While iterations stop condition not met
```

The local computation cost is on the order of O(|E| + |V|)/p, but with a large constant hidden by the O() notation. From experiments, the constant factor is about 10, considering the cost of sort or the random memory access penalty of hash table. The community assignment broadcast has a cost of |V| x 4 bytes, and the 'AllReduce' costs 2|V| x 8 bytes. These communication costs are the upper bound, assuming there are |V| communities, and all vertices update their community assignments. In practice, the communication cost can be much lower, depending on the dataset.

The graph contraction can be done as below on multiple GPUs:

```
temp_graph := Contract the local sub-graph based on community assignments;
Send edges <v, u, w> in temp_graph to host_GPU(v);
Merge received edges and form the new local sub-graph;
```

In the worst case, assuming the number of edges in the contracted graph is |E'|, the communication cost could be |E'| x 8 bytes, with the computation cost at about 5|E|/p + |E'|. For most datasets, the size reduction of graph from the contraction step is significant, so the memory needed to receive edges from the peer GPUs is manageable; however, if |E'| is large, it can be significant and may run out of GPU memory.

Summary of Louvain multi-GPU scaling

Because the modularity optimization runs multiple iterations before each graph contraction phase, the computation and communication of modularity optimization is dominant.

Parts	Comp cost	Comm cost	Comp/comm ratio	Scalability	Memory usage (B)
Modularity optim.	$10(E+V)/p$	$20V$ bytes	$E/p : 2V$	Okay	$88E/p + 12V$
Graph contraction	$5E/p + E'$	$8E'$ bytes	$5E/p + E' : 8E'$	Hard	$16E'$
Louvain	$10(E+V)/p$	$20V$ bytes	$E/p : 2V$	Okay	$88E/p + 12V + 16E'$

Louvain could be hard to implement on multiple GPUs, especially for the graph contraction phase, as it forms a new graph and distributes it across the GPUs. But the scalability should be okay.

GraphSAGE

The main memory usage and computation of SAGE are related to the features of vertices. While directly accessing the feature data of neighbors via peer access is possible and memory-efficient, it will create a huge amount of inter-GPU traffic that makes SAGE unscalable in terms of running time. Using UVM to store the feature data is also possible, but that will move the traffic from inter-GPU to the GPU-CPU connection, which is even less desirable. Although there is a risk of using up the GPU memory, especially on graphs that have high connectivity, a more scalable way is to duplicate the feature data of neighboring vertices. Depending on the graph size and the size of features, not all of the above data distribution schemes are applicable. The following analysis focuses on the feature duplication scheme, with other schemes' results in the summary table.

SAGE can be separated into three parts, depending on whether the computation and data access is source-centric or child-centric.

```
// Part1: Select the children
For each source in local batch:
    Select num_children_per_source children form source's neighbors;
    Send <source, child> pairs to host_GPU(child);

// Part2: Child-centric computation
For each received <source, child> pair:
    child_feature = feature[child];
    send child_feature to host_GPU(source);

    feature_sums := {0};
    For i from 1 to num_leaves_per_child:
        Select a leaf from child's neighbors;
        leaves_feature += feature[leaf];
    child_temp = L2_normalize( concatenate(
        feature[child] * Wf1, leaves_feature * Wa1));
    send child_temp to host_GPU(source);

// Part3: Source-centric computation
For each source in local batch:
    children_feature := sum(received child_feature);
    children_temp := sum(received child_temp);
    source_temp := L2_normalize( concatenate(
        feature[source] * Wf1, children_feature * Wa1));
```

```
source_result := L2_normalize( concatenate(
    source_temp * Wf2, children_temp * Wa2));
```

Assume the size of local batch is B, the number of children per source is C, the number of leaves per child is L, and the feature length per vertex is F. Dimensions of 2D matrices are noted as (x, y). The computation and communication costs for each part are:

```
Part 1, computation  : B x C.
Part 1, communication: B x C x 8 bytes.
Part 2, computation  : B x C x F + B x C x (F + L x F + F x (Wf1.y + Wa1.y)).
Part 2, communication: B x C x (F + Wf1.y + Wa1.y) x 4 bytes.
Part 3, computation  : B x (C x (F + Wf1.y + Wa1.y) + F x (Wf1.y + Wa1.y) +
                       (Wf1.y + Wa1.y) x (Wf2.y + Wa2.y)).
Part 3, communication: 0.
```

For Part 2's communication, if C is larger than about 2p, using `AllReduce` to sum up `child_feature` and `child_temp` for each source will cost less, at B x (F + Wf1.y + Wa1.y) x 2p x 4 bytes.

Summary of Graph SAGE multi-GPU scaling

Parts	Computation cost	Communication cost (Bytes)	Comp. to comm. ratio	Scalability
Feature duplication				
Children selection	BC	$8BC$	$1:8$	Poor
Child-centric comp.	$BCF \cdot (2 + L + \text{Wf1}.y + \text{Wa1}.y)$	$4B \cdot (F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p)$	$\sim CF : \min(C, 2p) \cdot 4$	Good
Source-centric comp.	$B \cdot (CF + (\text{Wf1}.y + \text{Wa1}.y) \cdot (C + F + \text{Wf2}.y + \text{Wa2}.y)$	0	N.A.	N.A.
Graph SAGE	$B \cdot (C + 3CF + 3LCF + (\text{Wf1}.y + \text{Wa1}.y) \cdot (CF + C + F + \text{Wf2}.y + \text{Wa2}.y))$	$8BC + 4B \cdot (F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p)$	at least $\sim CF : \min(C, 2p) \cdot 4$	Good

Parts	Computation cost	Communication cost (Bytes)	Comp. to comm. ratio	Scalability
Direct feature access				
Child-centric comp.	$BCF \cdot (2 + L + \text{Wf1}.y + \text{Wa1}.y)$	$4B \cdot ((F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p) + CLF)$	$\sim (2+L+\text{Wf1}.y+\text{Wa1}.y) : 4L$	poor
Graph SAGE	$B \cdot (C + 3CF + 3LCF + (\text{Wf1}.y + \text{Wa1}.y) \cdot (CF + C + F + \text{Wf2}.y + \text{Wa2}.y))$	$8BC + 4B \cdot (F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p) + 4BCFL$	$\sim (2+L+\text{Wf1}.y+\text{Wa1}.y) : 4L$	poor
Feature in UVM				
Child-centric comp.	$BCF \cdot (2 + L + \text{Wf1}.y + \text{Wa1}.y)$	$4B \cdot (F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p)$ bytes over GPU-GPU + $4BCLF$ bytes over GPU-CPU	$\sim (2 + L + \text{Wf1}.y + \text{a1}.y) : 4L$ over GPU-CPU	very poor
Graph SAGE	$B \cdot (C + 3CF + 3LCF + (\text{Wf1}.y + \text{Wa1}.y) \cdot (CF + C + F + \text{Wf2}.y + \text{Wa2}.y))$	$8BC + 4B \cdot (F + \text{Wf1}.y + \text{Wa1}.y) \cdot \min(C, 2p)$ bytes over GPU-GPU + $4BCFL$ bytes over GPU-CPU	$\sim (2+L+\text{Wf1}.y+\text{Wa1}.y) : 4L$ over GPU-CPU	very poor

When the number of features is at least several tens, the computation workload will be much more than communication, and SAGE should have good scalability. Implementation should be easy, as only simple p2p or AllReduce communication models are used. If memory usage is an issue, falling back to peer-access or UVM will result in very poor scalability; problem segmentation (i.e,. only process portion of the graph at a time) may be necessary to have a scalable implementation for large graphs, but that will be quite complex.

Random walks and graph search

If the graph can be duplicated on each GPU, the random walk multi-GPU 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.

A more interesting multi-GPU implementation would be when the graph is distributed across the GPUs. In this case, each step of a walk not only needs to send the `<walk#, step#, v>` information to `host_GPU(v)`, but also to the GPU that stores the result for such walk.

```
For each walk starting from local vertex v:
    Store v for ;
    Select a neighbor u of v;
    Store u for ;
    Send  to host_GPU(u) for visit;

Repeat until all steps of walks finished:
    For each received  for visit:
        Select a neighbor u of v;
        Send  to host_GPU(u) for visit;
        Send  to host_GPU_walk(walk#) for record;

    For each received  for record:
        Store v for ;
```

Using W as the number of walks, for each step, we have

Parts	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Random walk	W/p	$W/p * 24$ bytes	1 : 24	very poor

Graph search is very similar to random walk, except that instead of randomly selecting any neighbor, it selects the neighbor with the highest score (when using the `greedy` strategy), or with probabilities proportional to the neighbors' scores (when using the `stochastic_greedy` strategy). For the `greedy` strategy, a straightforward implementation, when reaching a vertex, goes through the whole neighbor list of thatsuch vertex and finds the one with maximum score. A more optimized implementation could perform a pre-visit to find the neighbor with maximum scored neighbor, with a cost of E/p; during the random walk process, the maximum scored neighbor will be known without going through the neighbor list.

For the `stochastic_greedy` strategy, the straightforward implementation would also go through all the neighbors, selecting one based on their scores and a random number. Preprocessing can also help: perform a scan on the scores of each vertex's neighbor list, with a cost of E/p; during the random walk, a binary search would be sufficient to select a neighbor, with weighted probabilities.

The cost analysis, depending on the walk strategy and optimization, results in:

Strategy	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Uniform	W/p	$W/p * 24$ bytes	$1:24$	Very poor
Greedy	Straightforward: dW/p	$W/p * 24$ bytes	$d:24$	Poor
Greedy	Pre-visit: W/p	$W/p * 24$ bytes	$1:24$	Very poor
Stochastic Greedy	Straightforward: dW/p	$W/p * 24$ bytes	$d:24$	Poor
Stochastic Greedy	Pre-visit: $log(d)W/p$	$W/p * 24$ bytes	$log(d):24$	Very poor

If the selection of a neighbor is weighted-random, instead of uniformly-random, it will increase the computation workload to Wd /p, where d is the average degree of vertices in the graph. As a result, the computation-to-communication ratio will increase to d:24; for most graphs, this is still not high enough to have good scalability.

Geolocation

In each iteration, Geolocation updates a vertex's location based on its neighbors. For multiple GPUs, neighboring vertices's location information needs to be available, either by direct access, UVM, or explicit data movement. The following shows how explicit data movement can be implemented.

```
Do
    Local geo location updates on local vertices;
    Broadcast local vertices' updates;
While no more update
```

The computation cost is on the order of O(|E|/p), if in each iteration all vertices are looking for possible location updates from neighbors. Because the spatial median function has a lot of mathematical computation inside, particularly a haversine() for each edge, the constant factor hidden by O() is large; for simplicity, 100 is used as the constant factor here. Assuming that we broadcast every vertex's location gives the upper bound of communication, but in reality, the communication should be much less, because 1) not every vertex updates its location every iteration and 2) vertices may not have neighbors on each GPU, so instead of broadcast, p2p communication may be used to reduce the communication cost, especially when the graph connectivity is low.

Comm. method	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Explicit movement	$100E/p$	$2V * 8$ bytes	$25E/p : 4V$	Okay
UVM or peer access	$100E/p$	$E/p * 8$ bytes	$25 : 1$	Good

Vertex Nomination

Vertex nomination is very similar to a single source shortest path (SSSP) problem, except it starts from a group of vertices, instead of a single source. One possible multi-GPU implementation is:

```
Set the starting vertex / vertices;
While has new distance updates
    For each local vertex v with distance update:
        For each edge <v, u, w> of vertex v:
            new_distance := distance[v] + w;
            if (distance[u] > new_distance)
                distance[u] = new_distance;

    For each u with distance update:
        Send <u, distance[u]> to host_GPU(u);
```

Assuming on average, each vertex has its distance updated `a` times, and the average degree of vertices is `d`, the computation and the communication costs are:

Parts	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Vertex nomination	aE/p	$aV/p * min(d, p) * 8$ bytes	$E : 8V * min(d, p)$	Okay

The min(d, p) part in the communication cost comes from update aggregation on each GPU: when a vertex has more than one distance update, only the smallest is sent out; a vertex that has a lot of neighbors and is connected to all GPUs has its communication cost capped by p x 8 bytes.

Scan Statistics

Scan statistics is essentially triangle counting (TC) for each vertex plus a simple post-processing step. The current Gunrock TC implementation is intersection-based: for an edge `<v, u>`, intersecting `neighbors[u]` and `neighbors[v]` gives

the number of triangles including `edge <v, u>`. This neighborhood-intersection-based algorithm only works if the neighborhood of end points of all edges for which we need to count triangles can reside in the memory of a single GPU. For graphs with low connectivities, such as road networks and meshes, it is still possible to partition the graph; for graphs with high connectivity, such as social networks or some web graphs, it's almost impossible to partition the graph, and any sizable partition of the graph may touch a large portion of vertices of the graph. As a result, for general graphs, the intersection-based algorithm requires the graph can be duplicated on each GPU. Under this condition, the multi-GPU implementation is trivial: only count triangles for a subset of edges on each GPU, and no communication is involved.

A more distributed-friendly TC algorithm is wedge-checking-based (https://e-reports-ext.llnl.gov/pdf/890544.pdf). The main idea is this: for each triangle A-B-C, where `degree(A) >= degree(B) >= degree(C)`, both A and B are in C's neighborhood, and A is in B's neighborhood; when testing for possible triangle D-E-F, with `degree(D) >= degree(E) >= degree(F)`, the wedge (two edges that share the same end point) that needs to be checked is D-E, and the checking can be simply done by verifying whether D is in E's neighborhood. As this algorithm is designed for distributed systems, it should be well-suited for multi-GPU system. The ordering requirements are imposed to reduce the number of wedge checks and to balance the workload. The multi-GPU pseudo code is:

```
For each local edge < v, u >:
    If (degree(v) > degree(u)) continue;
    For each neighbor w of v:
        If (degree(v) > degree(w)) continue;
        If (degree(u) > degree(w)) continue;
        Send tuple <u, w, v> to host_GPU(u) for checking;

For each received tuple < u, w, v >:
    If (w in u's neighbor list):
        triangles[u] ++;
        triangles[w] ++;
        triangles[v] ++;

AllReduce(triangles, sum);

// For Scan statistics only
For each vertex v in the graph:
    scan_stat[v] := triangles[v] + degree(v);
    if (scan_stat[v] > max_scan_stat):
        max_scan_stat := scan_stat[v];
        max_ss_node := v;
```

Using T as the number of triangles in the graph, the number of wedge checks is normally a few times `T`, noted as `aT`. For the three graphs tested by the LLNL paper—Twitter, WDC 2012, and Rmat-Scale 34—`a` ranges from 1.5 to 5. The large number of wedges can use up the GPU memory, if they are stored and communicated all at once. The solution is to generate a batch of wedges and check them, then generate another batch and check them, loop until all wedges are checked.

Assuming the neighbor lists of every vertex are sorted, the membership checking can be done in log(#neighbors). As a result, using `d` as the average outdegree of vertices, the cost analysis is:

Parts	Comp. cost	Comm. cost (B)	Comp/comm ratio	Scalability
Wedge generation	dE/p			
Wedge communication	0	$aE/p * 12$		
Wedge checking	$aE/p * log(d)$			
AllReduce	$2V$	$2V * 4$		
Triangle Counting	$(d * a * log(d))E/p + 2V$	$aE/p * 12 + 8V$	$\sim (d + a * log(d)) : 12a$	Okay
Scan Statistics (with wedge checks)	$(d * a * log(d))E/p + 2V + V/p$	$12aE/p + 8V$	$\sim (d + a * log(d)) : 12a$	Okay
Scan Statistics (with intersection)	$Vdd + V/p$	$8V$	$dd : 8$	Perfect

Sparse Fused Lasso (GTF)

The sparse fused lasso iteration is mainly a max-flow (MF), plus some per-vertex calculation to update the capacities in the graph. The reference and most non-parallel implementations of MF are augmenting-path-based; but finding the argumenting path and subsequent residual updates are both serial. The push-relabel algorithm is more parallelizable, and used by Gunrock's MF implementation. Each time the push operation updates the flow on an edge, it also needs to update the flow on the reverse edge; but the reverse edge may be hosted by another GPU, and that creates a large amount of inter-GPU traffic. The pseudocode for one iteration of MF with inter-GPU communication is:

```
// Push phase
```

```
For each local vertex v:
    If (excess[v] <= 0) continue;
    If (v == source || v == sink) continue;
    For each edge e<v, u> of v:
        If (capacity[e] <= flow[e]) continue;
        If (height[v] <= height[u]) continue;
        move := min(capacity[e] - flow[e], excess[v]);
        excess[v] -= move;
        flow[e] += move;
        Send <reverse[e], move> to host_GPU(u);
        If (excess[v] <= 0)
            break for each e loop;

For each received <e, move> pair:
    flow[e] -= move;
    excess[Dest(e)] += move;

// Relabel phase
For each local vertex v:
    If (excess[v] <= 0) continue;
    min_height := infinity;
    For each e<v, u> of v:
        If (capacity[e] <= flow[e]) continue;
        If (min_height > height[u])
            min_height = height[u];
    If (height[v] <= min_height)
        height[v] := min_height + 1;

Broadcast height[v] for all local vertex;
```

The cost analysis will not be on one single iteration, but on a full run of the push-relabel algorithm, as the bounds of the push and the relabel operations are known.

Parts	Comp. cost	Comm. cost (Bytes)	Comp/comm ratio	Scalability
Push	$a(V+1)VE/p$	$(V+1)VE/p*8$	$a:8$	Less than okay
Relabel	VE/p	V^2*8	$d/p:8$	Okay
MF (Push-Relabel)	$(aV+a+1)VE/p$	$V^2((V+1)d/p+1)*8$	$\sim a:8$	Less than okay

The GTF-specific parts are more complicated than MF in terms of communication: the implementation must keep some data, such as weights and sizes, for each community of vertices, and multiple GPUs could be updating such data simultaneously. It's almost impossible to do explicit data movement for this part, and the best option is to use direct access or UVM; each vertex may update its community once, so the communication cost is still manageable. One iteration of GTF is:

```
MF;
BFS to find the min-cut;
Vertex-Community updates;
Updates source-vertex and vertex-destination capacities;
```

with scaling characteristics:

Parts	Comp. cost	Comm. cost (Bytes)	Comp/comm ratio	Scalability
MF (Push-Relabel)	$(aV + a + 1)VE/p$	$V^2((V + 1)d/p + 1) * 8$	$\sim a : 8$	Less than okay
BFS	E/p	$2V * 4$	$d/p : 8$	Okay
V-C updates	E/p	$V/p * 8$	$d : 8$	Okay
Capacity updates	V/p	$V/p * 4$	$1 : 4$	Less than okay
GTF	$(aV + a + 1)VE/p$ $+2E/p+V/p$	$V^2((V + 1)d/p + 1) * 8$ $+2Vx4+V/p*4$	$\sim a : 8$	Less than okay

It's unsurprising that GTF may not scale: the compute- and communicate-heavy part of GTF is the MF, and in MF, each push needs communication to update its reverse edge. A more distributed-friendly MF algorithm is needed to overcome this problem.

Graph Projection

Graph projection is very similar to triangle counting by wedge checking; but instead of counting the triangles, it actually records the wedges. The problem here is not computation or communication, but rather the memory requirement of the result: projecting all vertices may create a very dense graph, which may be much larger than the original graph. One possible solution is to process the results in batches:

```
vertex_start := 0;
While (vertex_start < num_vertices)
    markers := {0};
    current_range := [vertex_start, vertex_start + batch_size);
    For each local edge e<v, u> with u in current_range:
        For each neighbor t of v:
            If (u == t) continue;
            markers[(u - vertex_start) * ceil(num_vertices / 32) + t / 32] |=
                1 << (t % 32);

    For each vertex u in current_range:
        Form the neighbor list of u in the new graph by markers;

    For each local edge e<v, u> with u in current_range:
        For each neighbor t of v:
            If (u == t) continue;
            e' := edge_id of <u, t> in the new graph,
                by searching u's neighbor list;
            edge_values[e'] += 1;

    For each edge e'<u, t, w> in the new graph:
        send <u, t, w> to host_GPU(u);

    Merge all received <u, t, w> to form projection
        for local vertices u in current_range;
    Move the result from GPU to CPU;

    vertex_start += batch_size;
```

Using `E'` to denote the number of edges in the projected graph, and `d` to denote the average degree of vertices, the costs are:

Parts	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Marking	dE/p	0 byte		
Forming edge lists	E'	0 byte		
Counting	dE/p	0 byte		
Merging	E'	$E' * 12$ bytes		
Graph Projection	$2dE/p + 2E'$	$12E'$ bytes	$dE/p + E' : 6E'$	Okay

If the graph can be duplicated on each GPU, instead of processing distributed edges, each GPU can process only `u` vertices that are hosted on that GPU. This

eliminates the merging step; as a result, there is no communication needed, and the computation cost reduces to 2dE/p + E'.

Local Graph Clustering

The Gunrock implementation of Local Graph Clustering (LGC) uses PageRank-Nibble, a variant of the PageRank algorithm. PR-Nibble's communication pattern is the same as standard PR: accumulate changes for each vertex to its host GPU. As a result, PR-Nibble should be scalable, just as standard PR. PR-Nibble with communication can be done as:

```
// Per-vertex updates
For each active local vertex v:
    If (iteration == 0 && v == src\_neighbor) continue;
    If (iteration > 0 && v == src)
        gradient[v] -= alpha / #reference_vertices / sqrt(degree(v));
    z[v] := y[v] - gradient[v];
    If (z[v] == 0) continue;

    q_old := q[v];
    threshold := rho * alpha * sqrt(degree(v));
    If (z[v] >= threshold)
        q[v] := z[v] - threshold;
    Else if (z[v] <= -threshold)
        q[v] := z[v] + threshold;
    Else
        q[v] := 0;

    If (iteration == 0)
        y[v] := q[v];
    Else
        y[v] := q[v] + (1 - sqrt(alpha)) / (1 + sqrt(alpha)) * (q[v] - old_q);

    gradient[v] := y[v] * (1 + alpha) / 2;

// Ranking propagation
For each edge e<v, u> of active local vertex v:
    change := y[v] * (1 - alpha) / 2 / sqrt(degree(v)) / sqrt(degree(u));
    gradient_update[u] -= change;

For each u that has gradient updates:
    send < u, gradient_update[u]> to host_GPU(u);

For each received gradient update < u, gradient_update>:
    gradient[u] += gradient_update;
```

```
// Gradient updates
For each local vertex u with gradient updated:
    If (gradient[u] == 0) continue;
    Set u as active for next iteration;

    val := gradient[u];
    If (u == src)
        val -= (alpha / #reference_vertices) / sqrt(degree(u));
    val := abs(val / sqrt(degree(u)));
    if (gradient_scale_value < val)
        gradient_scale_value = val;
    if (val > gradient_threshold)
        gradient_scale := 1;
```

The cost analysis is:

Parts	Comp. cost	Comm. cost	Comp/comm ratio	Scalability
Per-vertex updates	$\sim 10V/p$	0 bytes		
Ranking propagation	$2E/p$	$V * 8$ bytes	$d/p : 4$	
Gradient updates	V/p	0 bytes		
Local graph clustering	$(12V + 2E)/p$	$8V$ bytes	$(6 + d)/p : 4$	good

Seeded Graph Matching and Application Classification

The implementations of these two applications are linear-algebra-based, as opposed to other applications where we used Gunrock and its native graph (vertex-edge) data structures. The linear-algebra-based (BLAS-based) formulations, especially the ones that require matrix-matrix multiplications, may impose a large communication requirement. Advance matrix-matrix and vector-matrix multiplication kernels use optimizations that build on top of a specific layout of the data, which may be not distribution-friendly. A different method of analyzing the computation and the communication costs—the computation vs. communication ratio—is needed for these applications.

Summary of Results

Application	Computation to communication ratio	Scalability	Implementation difficulty
Louvain	$E/p : 2V$	Okay	Hard
Graph SAGE	$\sim CF : \min(C, 2p) \cdot 4$	Good	Easy

Application	Computation to communication ratio	Scalability	Implementation difficulty
Random walk	Duplicated graph: infinity Distributed graph: $1:24$	Perfect Very poor	Trivial Easy
Graph search: Uniform	$1:24$	Very poor	Easy
Graph search: Greedy	Straightforward: $d:24$ Pre-visit: $1:24$	Poor Very poor	Easy Easy
Graph search: Stochastic greedy	Straightforward: $d:24$ Pre-visit: $\log(d):24$	Poor Very Poor	Easy Easy
Geolocation	Explicit movement: $25E/p:4V$ UVM or peer access: $25:1$	Okay Good	Easy Easy
Vertex nomination	$E:8V\cdot\min(d,p)$	Okay	Easy
Scan statistics	Duplicated graph: infinity Distributed graph: $\sim(d+a\cdot\log(d)):12$	Perfect Okay	Trivial Easy
Sparse fused lasso	$\sim a:8$	Less than okay	Hard
Graph projection	Duplicated graph : infinity Distributed graph : $dE/p+E':6E'$	Perfect Okay	Easy Easy
Local graph clustering	$(6+d)/p:4$	Good	Easy
Seeded graph matching			
Application classification			

Seeded graph matching and application classification are matrix-operation-based and not covered in this table.

From the scaling analysis, we can see these workflows can be roughly grouped into three categories, by their scalabilities:

Good scalability GraphSAGE, geolocation using UVM or peer accesses, and local graph clustering belong to this group. They share some algorithmic signatures: the whole graph needs to be visited at least once in every iteration, and visiting each edge involves nontrivial computation. The communication costs are roughly at the level of V. As a result, the computation vs. communication ratio is larger than E : V. PageRank is a standard graph algorithm that falls in this group.

Moderate scalability This group includes Louvain, geolocation using explicit movement, vertex nomination, scan statistics, and graph projection. They either only visit part of the graph in an iteration, have only trivial computation during an edge visit, or communicate a little more data than V. The computation

vs. communication is less than E : V, but still larger than 1 (or 1 operation : 4 bytes). They are still scalable on the DGX-1 system, but not as well as the previous group. Single source shortest path (SSSP) is an typical example for this group.

Poor scalability Random walk, graph search, and sparse fused lasso belong to this group. They need to send out some data for each vertex or edge visit. As a result, the computation vs communication ratio is less than 1 (or 1 operation : 4 bytes). They are very hard to scale across multiple GPUs. Random walk is an typical example.

End of report