AFRL-RY-WP-TR-2021-0135

A COMMODITY PERFORMANCE BASELINE FOR HIERARCHICAL IDENTIFY VERIFY EXPLOIT (HIVE) GRAPH APPLICATIONS

Ben Johnson, Muhammad Osama, John D. Owens, and Serban Porumbescu
University of California Davis

SEPTEMBER 2021
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

AFRL-RY-WP-TR-2021-0135 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

DAVID J. LUCKING
Program Manager
Sensors Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

TIMOTHY R. JOHNSON, Chief
Sensors Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

LESTER C. LONG, Lt Col, USAF
Deputy Chief
Aerospace Components & Subsystems Division
Sensors Directorate

haveREPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE ***(DD-MM-YY)*** September 2021	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 8 June 2018 – 28 February 2021

4. TITLE AND SUBTITLE A COMMODITY PERFORMANCE BASELINE FOR HIERARCHICAL IDENTIFY VERIFY EXPLOIT (HIVE) GRAPH APPLICATIONS	**5a. CONTRACT NUMBER** FA8650-18-2-7835
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Ben Johnson, Muhammad Osama, John D. Owens, and Serban Porumbescu	**5d. PROJECT NUMBER** N/A
	5e. TASK NUMBER N/A
	5f. WORK UNIT NUMBER Y1SM
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California Davis 1 Shields Ave. Davis, CA 95616	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Force; Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RYDR
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-RY-WP-TR-2021-0135

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This material is based on research sponsored by the Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7835. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Labs (AFRL), the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. Report contains color.

14. ABSTRACT
Herein UC Davis produces the following deliverables that it promised to deliver in Phase 2:
- Implementation of DARPA HIVE v0 apps as single-node, multi-GPU applications using the Gunrock framework
- Performance characterization of these applications across multiple GPUs
- Analysis of the limits of scalability for these applications

In our writeup, we first describe how to reproduce our results and then describe the scalability behavior of our ForAll operator.

15. SUBJECT TERMS
HIVE, graph, GPU, parallel, Gunrock

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**8. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	SAR	90	David Lucking **19b. TELEPHONE NUMBER** *(Include Area Code)* N/A

Table of Contents

Section Page

Section Page

Section Page

Section Page

List of Figures

List of Tables

Table **Page**

1 HIVE PHASE 2 REPORT: EXECUTIVE SUMMARY

This report is also located online at the following URL: **https://gunrock.github.io/docs/#/hive_phase2/hive_phase2_summary**. Links currently work better in the PDF version than the HTML version.

Herein UC Davis produces the following deliverables that it promised to deliver in Phase 2:

- Implementation of DARPA HIVE v0 apps as single-node, multi-GPU applications using the **Gunrock** framework
- Performance characterization of these applications across multiple GPUs
- Analysis of the limits of scalability for these applications

In our writeup, we first **describe how to reproduce our results** (**HTML**) and then **describe the scalability behavior of our ForAll operator** (**HTML**) .

We begin with a table that summarizes the scalability behavior for each application, then a longer description of each application:

Application	Scalability behavior
Scan Statistics	Bottlenecked by single-GPU and communication
GraphSAGE	Bottlenecked by network bandwidth between GPUs
Application Classification	Bottlenecked by network bandwidth between GPUs
Geolocation	Bottlenecked by network bandwidth between GPUs
Community Detection (Louvain)	Application is nonfunctional
Local Graph Clustering (LGC)	Bottlenecked by single-GPU and communication
Graph Projections	Limited by load imbalance
GraphSearch	Bottlenecked by network bandwidth between GPUs
Seeded Graph Matching (SGM)	We observe great scaling
Sparse Fused Lasso	Maxflow kernel is serial
Vertex Nomination	We observe weak scaling

1.1 App: Scan Statistics (HTML)

We rely on Gunrock's multi-GPU `ForALL` operator to implement Scan Statistics. We see no scaling and in general performance degrades as we sweep from one to sixteen GPUs. The application is likely bottlenecked by the single GPU intersection operator that requires a two-hop neighborhood lookup and accessing an array distributed across multiple GPUs.

1.2 App: GraphSAGE (HTML)

We rely on Gunrock's multi-GPU `ForALL` operator to implement GraphSAGE. We see no scaling as we sweep from one to sixteen GPUs due to communication over GPU interconnects.

1.3 App: Application Classification **(**HTML**)**

We re-forumlate the **application_classification** workload to improve memory locality and admit a natural multi-GPU implementation. We then parallelized the core computational region of **application_classification** across GPUs. For the kernels in that region that do not require communication between GPUs, we attain near-perfect scaling. Runtime of the entire application remains bottlenecked by network bandwidth between GPUs. However, mitigating this bottleneck should be possible further optimization of the memory layout.

1.4 App: Geolocation **(**HTML**)**

We rely on Gunrock's multi-GPU **ForALL** operator to implement Geolocation as the entire behavior can be described within a single-loop like structure. The core computation focuses on calculating a spatial median, and for multi-GPU **ForAll**, that work is split such that each GPU gets an equal number of vertices to process. We see a minor speed-up on a DGX-A100 going from 1 to 3 GPUs on a twitter dataset, but in general, due to the communication over the GPU-GPU interconnects for all the neighbors of each vertex, there's a general pattern of slowdown going from 1 GPU to multiple GPUs, and no scaling is observed.

1.5 App: Community Detection (Louvain) **(**HTML**)**

The application has a segmentation fault and is currently nonfunctional.

1.6 App: Local Graph Clustering (LGC) **(**HTML**)**

We rely on Gunrock's multi-GPU `ForALL` operator to implement Local Graph Clustering and observe no scaling as we increase from one to sixteen GPUs. The application is likely bottlenecked by single-GPU filter and advance operators and communication across NVLink necessary to access arrays distributed across GPUs.

1.7 App: Graph Projections **(**HTML**)**

We implemented a multi-GPU version of sparse-sparse matrix multiplication, based on chunking the rows of the left hand matrix. This yields a communication-free implementation with good scaling properties. However, our current implementation remains partially limited by load imbalance across GPUs.

1.8 App: GraphSearch **(**HTML**)**

We rely on a Gunrock's multi-GPU `ForALL` operator to implement GraphSearch as the entire behavior can be described within a single-loop like structure. The core computation focuses on determining which neighbor to visit next based on uniform, greedy, or stochastic functions. Each GPU is given an equal number of vertices to process. No scaling is observed, and in general we see a pattern of decreased performance as we move from 1 to 16 GPUs due to random neighbor access across GPU interconnects.

1.9 App: Seeded Graph Matching (SGM) **(**HTML**)**

Multi-GPU SGM experiences considerable speed-ups over single GPU implementation with a near linear scaling if the dataset being processed is large enough to fill up the GPU. We notice that ~1 million nonzeros sparse-matrix is a decent enough size for us to show decent scaling as we increase the number of GPUs. The misalignment for this implementation is also synthetically generated (just like it was for Phase 1, the bottleneck is still the **|V|x|V|** allocation size).

1.10 App: Sparse Fused Lasso **(**HTML**)**

Sparse Fused Lasso (or Sparse Graph Trend Filtering) relies on a Maxflow algorithm. As highlighted in the Phase 1 report, a sequential implementation of Maxflow outperforms a single-GPU implementation, and the actual significant core operation of SFL is a serial normalization step that cannot be parallelized to a single GPU, let alone multiple GPUs. Therefore, we refer readers to the phase 1 report for this workload. Parallelizing across multiple GPUs is not beneficial.

1.11 App: Vertex Nomination **(**HTML**)**

We implemented **vertex_nomination** as a standalone CUDA program, and achieve good weak scaling performance by eliminating communication during the **advance** phase of the algorithm and using a frontier representation that allows an easy-to-compute reduction across devices. We also produce web versions of our **scalability plots** and **scalability tables of results**.

2 RUNNING THE APPLICATIONS

Given the number of applications, options, datasets, and GPU configurations, we have tried to simplify application testing as much as possible. To facilitate test sweeps across 1 to 16 GPUs, multiple application options, and datasets, every application has two associated scripts: **hive-mgpu-run.sh** and **hive-application-test.sh**.

In general **hive-mgpu-run.sh** deals with parameter sweeps and schedules **hive-application-test.sh** as multi-GPU SLURM jobs. The **hive-application-test.sh** script generally deals with datasets and associated paths, and configures itself with the parameters necessary to run the application.

2.1.1 Default Run Configuration

The simplest way to run an application is to execute:
./hive-mgpu-run.sh
The associated **hive-application-test.sh** will execute with datasets in the following user directories on NVIDIA's **nslb** cluster:
/home/u00u7u37rw7AjJoA4e357/data/gunrock/gunrock_dataset
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets

2.1.2 Alternate Run Configurations

Additional command line parameters and / or script modifications are necessary to run on additional datasets or with alternate application parameters.

2.1.2.1 hive-mgpu-run.sh

This script configures SLURM with **NUM_GPUS** to sweep across on a chosen **PARTITION_NAME**. Running the script with no parameters (as shown above) is equivalent to:
./hive-mgpu-run.sh 16 dgx2

This runs **hive-application-test.sh** across 1 to 16 GPUs on the machine partition named **dgx2**.

For some applications, this script might have additional parameter variables that are worth exploring and modifying. Please see the individual HIVE application chapters for more details.

2.1.2.2 hive-application-test.sh

The primary reason to modify this script is to provide additional dataset information. In general these scripts will include some or all of the following arrays:

- **DATA_PREFIX** path to directory containing desired dataset
- **NAME** a simple string naming the dataset, generally sans a file extension (e.g., **NAME[0]="twitter"** for **twitter.mtx**)
- **GRAPH** aggregated options for the chosen dataset to pass to the application (i.e., combine **DATA_PREFIX** and **NAME** with additional information expected by the application)

Please note that you must update the associated for loop index if you add or remove items to the arrays mentioned.

2.1.3 Future Script Simplification

In the future we would like to refactor **hive-mgpu-run.sh** to simply configure the necessary SLURM command (e.g., resources and hardware partition) and pass the command to the **hive-application-test.sh** script. The application script can then deal with sweeping across its relevant parameters and datasets.

3 GUNROCK'S FORALL OPERATOR

Gunrock's `ForAll` operator is a `compute` operator type, meaning, it takes in an input array and applies user-defined function on every element of the input in **parallel**. This input for the `ForAll` operator can in a sense be any element of an array, vertices or edges of a frontier, or all the vertices or edges of the entire graph. In HIVE's phase I, due to the intuitive nature and simple implementation of the the parallel `ForAll` operator, we found that the operator was very useful in implementing single-GPU versions of several of the HIVE application workloads such as Geolocation, Graph Search, Random Walk, GraphSAGE, computation elements of Local Graph Clustering, Louvain, and Graph Trend Filtering.

The following pseudocode shows a simple-sequential implementation of the `ForAll` operator:

```
template <typename ArrayT, typename ApplyLambda>
void ForAll(ArrayT* array, ApplyLambda apply, std::size_t size) {
  for(std::size_t i = 0; i < size; ++i)
    apply(array, i);
}
```

3.1 Summary of Multi-GPU `ForAll`

In this write-up, we show how Gunrock's **ForAll** operator can be extended to support multiple GPU execution. We also explain what kind of scaling is expected with the new multi-GPU **ForAll** versus the kind of scaling we observe in real-world problems (such as the HIVE applications). We elaborate on what the performance bottlenecks are for our current implementation and what can we do better in the future with specialized-scalable operators targetting interesting patterns present in these applications.

3.2 `ForAll` Implementation

3.2.1 Approach to Single-GPU

CUDA-based implementation of a parallel `ForAll` operator is a simple extension to the sequential version described above, where instead of looping over the array in a sequential loop, we launch `ceil_div(size, BLOCK_SIZE)` blocks, with 128 or 256 threads per block, and each element of the array gets processed by each thread of the parallel CUDA grid launch. This effectively makes a simple loop-operator, a parallel operator with the ability to apply any arbitrary user-defined operator on every element of the array. Given a single-GPU parallel `ForAll` operator, the users working on the graph algorithms can then write their custom user-defined operators to implement the apps. One example of an application implemented entirely using `ForAll` is Geolocation (described in detail **here**). The following snippet is the CUDA-kernel call for Gunrock's `ForAll` operator.

```
template <typename ArrayT, typename ApplyLambda>
__global__ void ForAll_Kernel(ArrayT array, ApplyLambda apply, std::size_t
size) {
  const std::size_t STRIDE = blockDim.x * gridDim.x;
  auto thread_idx = blockDim.x * blockIdx.x + threadIdx.x;
  while (thread_idx < size) {
    apply(array, thread_idx);
    i += STRIDE;
  }
}
```

3.2.2 Approach to Multi-GPU

Extending Gunrock's **`ForAll`** operator from single-GPU to multiple GPUs can be achieved by using CUDA's multi-stream model. A *stream* in CUDA programming model introduces asynchrony such that independent tasks (or kernels) can run concurrently. If, no stream is specified, CUDA assumes the kernels are all running under a special **default** stream called the **`NULL`** stream. **`NULL`** stream's behavior is such that each task on the **`NULL`** stream synchronizes before running the next task, effectively making it sequential. However, it is important for multiple GPU streams to all execute in parallel, therefore, we create a stream for each GPU and a "master" stream, which every stream synchronizes to at the very end to signal that the task has been completed.

The following simplified snippet shows how one can create, launch and synchronize a stream per GPU for the **`ForAll`** operator:

```
// Create a stream per GPU:
std::vector<cudaStream_t> streams(num_gpus);
for(int i = 0; i < num_gpus; ++i) {
  cudaSetDevice(i);
  cudaStreamCreate(&streams[i]);
}

// Launch kernels on individual streams:
for(int i = 0; i < num_gpus; ++i) {
  cudaSetDevice(i);
  ForAll<<<GRID_DIM, BLOCK_DIM, 0, streams[i]>>>(...);
}

// Synchronize each streams:
for(int i = 0; i < num_gpus; ++i) {
  cudaSetDevice(i);
  cudaStreamSynchronize(streams[i]);
}
```

The above is a great initial formulation to achieve asynchronous *device-side* launch of our **`ForAll`** kernel, but we can do better! Even though the device-side execution is now asynchronous with the multi-streams abstraction, on the CPU-side, we are still launching kernels sequentially. We can remedy that by using muiltiple CPU threads to asynchronously launch our kernels on multiple streams from the CPU using OpenMP or C++ threads:

```
// We can use openmp or C++ thread to achieve the multithreaded launch:
#pragma omp parallel for
for(int i = 0; i < num_gpus; ++i) {
  cudaSetDevice(i);
  ForAll<<<GRID_DIM, BLOCK_DIM, 0, streams[i]>>>(...);
}
```

Now, to be able to actually work on individual data elements per GPU, we simply offset the input array by **`gpu_id * (size / num_gpus)`**, such that each GPU gets a unique section of the work to process.

3.3 Scalability Analysis

3.3.1 Expected vs. Observed Scaling

Multiple GPUs **`ForAll`** operator was initially intended as a **`transform`** operator, where given an array we apply a user-defined transformation on every element of the array. If the user-defined operations are restricted to the array/elements being processed and are simple, the observed scaling is linear. Each GPU gets an embarassingly parallel chunk of work to do independent of every other GPU on the system, therefore, expected scaling to be perfect-linear.

However, what we observe in practice is that the user-defined functions can be complex computations used to implement some of the HIVE workloads. An example pattern that the user may want can be described as following:

1. "Array" being processed in the **`ForAll`** is an active vertex set of the graph,
2. Therefore, giving access to each vertex in a frontier within the user-defined operation,
3. And in the operation itself, the user may do any random access to other arrays in the algorithm's problem.

These random accesses are observed in many applications, for example, in Geolocation you may want to get the latitude and longitude for each vertex in the graph, and get the latitude and longitude of each of the neighbors of that given vertex to find a spatial-distance. In an ideal case, the neighbor's vertices data is local to each GPU, but in practice, that neighbor could live in any of the GPUs in a system, which causes the GPU processing the neighbor, to incur remote memory transaction causing our expected perfectly linear scaling to fail.

3.3.2 Performance Limitations

For our multi-GPU work, we deploy three different memory schemes for allocating/managing the data that gets split equally among all the GPUs in the systems, these schemes are:

1. Unified Memory Addressing Space (**`cudaEnablePeerAccess()`**)

2. Unified Memory Management (**`cudaMallocManaged()`**)
3. CUDA Virtual Memory Management (**`cuMemAddressReserve()`**)

With the help of prefetching the managed memory (2), all three APIs can perform nearly the same and achieve the goal of allowing all the data within Gunrock to be accessed by all of the GPUs in the system. One additional optimization we deploy is replicating the input graph to all GPUs if the graph is small enough to fit in a single-GPU memory space. As hinted earlier, the performance bottleneck is not within these memory schemes, as they simply split the data and allow it to be accessed from a unified address space, it is within the memory accesses that the user defines within the compute lambda. When lots of remote memory accesses occur from a single-GPU, it can saturate the memory bus causing operations to halt until these transactions are completed. This makes it so that the problems take longer to solve in a multiple GPU system versus a single GPU, because many of the computations are waiting on memory to arrive from remote GPUs. The problem can be reduced by using faster interconnects, such as the new NVLink in the Ampere A100s, but due to Ampere A100s having more compute units as well, the interconnects' bandwidth is still not enough to saturate the device.

We found that although there are some accesses that are entirely random, many of the user-defined lambdas can be split into multiple parts and common patterns can be further extracted into operators. Once we switch to this specialized-operator model, we can scale our problems better (as further explained in the following section).

3.3.3 Optimizations and Future Work

One lesson learned from implementing a multiple GPU **`ForAll`** operator is that there is a need to identify common patterns within the **`ForAll`** user-defined implementations to be made into operators that can potentially scale. Continuing the previously mentioned Geolocation example, we can look into implementing Geolocation with **`NeighborReduction`**, where **`Reduction`** is not a simple reduce, but more complex user-defined operations (such as **`spatial-median`**). Another reason why moving onto specialized graph operators instead of a general **`ForAll`** will be better is that we can then map communication patterns within these operators to be able to transfer information at a per-iteration basis between different GPUs using gather, scatter, broadcast (can be achieved using **NCCL** primitives.) We show one such example with Vertex Nomination, implemented using NCCL, an NVIDIA communication library for multiple GPUs.

4 SCAN STATISTICS

From the **Phase 1 report** for Scan Statistics:

Scan statistics, as described in **Priebe et al.**, is the generic method that computes a statistic for the neighborhood of each node in the graph, and looks for anomalies in those statistics. In this workflow, we implement a specific version of scan statistics where we compute the number of edges in the subgraph induced by the one-hop neighborhood of each node u in the graph. It turns out that this statistic is equal to the number of triangles that node u participates in plus the degree of u. Thus, we are able to implement scan statistics by making relatively minor modifications to our existing Gunrock triangle counting (TC) application.

4.1 Scalability Summary

Bottlenecked by single-GPU and communication

4.2 Summary of Results

We rely on Gunrock's multi-GPU `ForALL` operator to implement Scan Statistics. We see no scaling and in general performance degrades as we sweep from one to sixteen GPUs. The application is likely bottlenecked by the single GPU intersection operator that requires a two-hop neighborhood lookup and accessing an array distributed across multiple GPUs.

4.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.

We parallelize Scan Statistics by utilizing a multi-GPU `ForAll` operator that splits the `scan_stats` array evenly across all available GPUs. Additional information on multi-GPU `ForAll` can be found in **Gunrock's `ForAll` Operator** section of the report. Furthermore, this application depends on triangle counting and an intersection operator that have not been parallelized (i.e., across multiple GPUs). It is not clear that simply parallelizing these functions would lead to scalability due to the communication patterns they exhibit.

4.3.1 Differences in Implementation from PHASE 1

No change from Phase 1.

4.4 How to Run this Application on NVIDIA's DGX-2

4.4.1 Prerequisites

```
git clone  https://github.com/gunrock/gunrock -b multigpu
mkdir build
cd build/
cmake ..
make -j16 ss
```
Verify git SHA: `commit d70a73c5167c5b59481d8ab07c98b376e77466cc`

4.4.2 Partitioning the Input Dataset

Partitioning is handled automatically as Scan Statistics relies on Gunrock's multi-GPU **`ForALL`** operator and its **`scan_stats`** array is split evenly across all available GPUs (see **`ForAll`** for details).

4.4.3 Running the Application (Default Configurations)

From the **`build`** directory

```
cd ../examples/ss/
./hive-mgpu-run.sh
```

This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application option as specified in **`hive-ss-test.sh`**. See **Running the Applications** for more details.

4.4.3.1 Datasets

Default Locations:
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-2TB
Names:
pokec

4.4.4 Running the Application (alternate configurations)

4.4.4.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

4.4.4.2 hive-ss-test.sh

Modify **APP_OPTIONS** to specify alternate **--undirected** and **--num-runs** values. Please see the Phase 1 single-GPU implementation details **here** for additional parameter information.

Please review the provided script and see "Running the Applications" chapter for details on running with additional datasets.

4.4.5 Output

No change from Phase 1.

4.5 Performance and Analysis

No change from Phase 1.

4.5.1 Implementation limitations

No change from Phase 1.

4.5.2 Performance Limitations

Single-GPU: No change from Phase 1.

Multiple-GPUs: Performance bottleneck is likely the single-GPU implementation of triangle counting and intersection and the need to randomly access an array distributed across multiple GPUs. Though once parallelized across multiple GPUs, the random access patterns of these functions (e.g., two-hop neighborhoods) would bottleneck communication over NVLink.

4.6 Scalability Behavior

We observe no scaling with the current Scan Statistics implementation. Please see the chapter on **Gunrock's ForAll Operator** for a discussion on future directions around more specialized operators to be designed with communication patterns in mind.

4.7 Scalability Plots

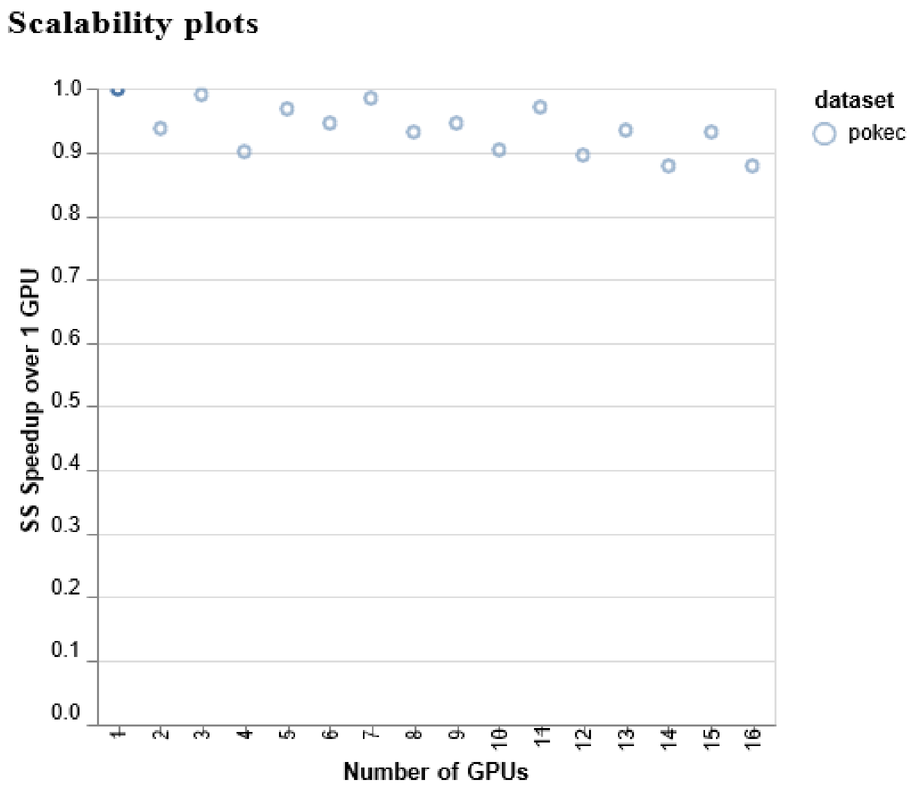

Figure 1: SS: Speedup over 1 GPU vs. Number of GPUs

5 GRAPHSAGE

The **Phase 1 writeup** contains a detailed description of the application.

From the Phase 1 writeup:

GraphSAGE is a way to fit graphs into a neural network: instead of getting the embedding of a vertex from all its neighbors' features as in conventional implementations, GraphSAGE selects some 1-hop neighbors, some 2-hop neighbors connected to those 1-hop neighbors, and computes the embedding based on the features of the 1-hop and 2-hop neighbors. The embedding can be considered as a vector containing hash values describing the interesting properties of a vertex.

5.1 Scalability Summary

Bottlenecked by network bandwidth between GPUs

5.2 Summary of Results

We rely on Gunrock's multi-GPU `ForALL` operator to implement GraphSAGE. We see no scaling as we sweep from one to sixteen GPUs due to communication over GPU interconnects.

5.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.

We parallelize across GPUs by utilizing a multi-GPU `For-All` operator and evenly distribute relevant arrays across multiple GPUs. Please see **Gunrock's `ForAll` Operator** for more details.

5.3.1 Differences in Implementation from Phase 1

no change from phase 1.

5.4 How to Run this application on NVIDIA's DGX-2 Prerequisites

```
git clone  https://github.com/gunrock/gunrock -b multigpu
mkdir build
cd build/
cmake ..
make -j16 sage
```

Verify git SHA: `commit d70a73c5167c5b59481d8ab07c98b376e77466cc`

5.4.1 Partitioning the Input Dataset

Partitioning is handled automatically as GraphSage relies on Gunrock's multi-GPU `ForALL` operator and its frontier vertices are split evenly across all available GPUs. Please refer to the chapter on **Gunrock's `ForAll` Operator** for additional information.

5.4.2 Running the Application (Default Configurations)

From the `build` directory
```
cd ../examples/sage/
./hive-mgpu-run.sh
```
This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application option as specified in **hive-sage-test.sh**.

Running the Applications chapter for details on running with additional datasets for additional parameter information, review the provided script, and see **Running the Applications** chapter for details on running with additional datasets.

5.4.2.1 Datasets

Default Locations:
```
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-2TB
/home/u00u7u37rw7AjJoA4e357/data/gunrock/gunrock_dataset/mario-2TB/large
```
Names:
```
pokec
dir_gs_twitter
europe_osm
```

5.4.3 Running the Application (Alternate Configurations)

5.4.3.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

5.4.3.2 hive-sage-test.sh

Modify **APP_OPTIONS** to specify alternate **--undirected** and **--batch-size** options. Please see the Phase 1 single-GPU implementation details **here** for additional parameter information, review the provided script, and see **Running the Applications** chapter for details on running with additional datasets.

5.4.4 Output

No change from Phase 1.

5.5 Performance and Analysis

No change from Phase 1.

5.5.1 Implementation Limitations

No change from Phase 1.

5.5.2 Performance Limitations

Single-GPU: No change from Phase 1.
Multiple-GPUs: Performance bottleneck is the remote memory accesses from one GPU to another GPU's memory through NVLink.

5.6 Scalability Behavior

We observe no scaling with the current GraphSAGE implementation. Please see the chapter on **Gunrock's ForAll Operator** for a discussion on future directions around more specialized operators to be designed with communication patterns in mind.

5.7 Scalability Plots

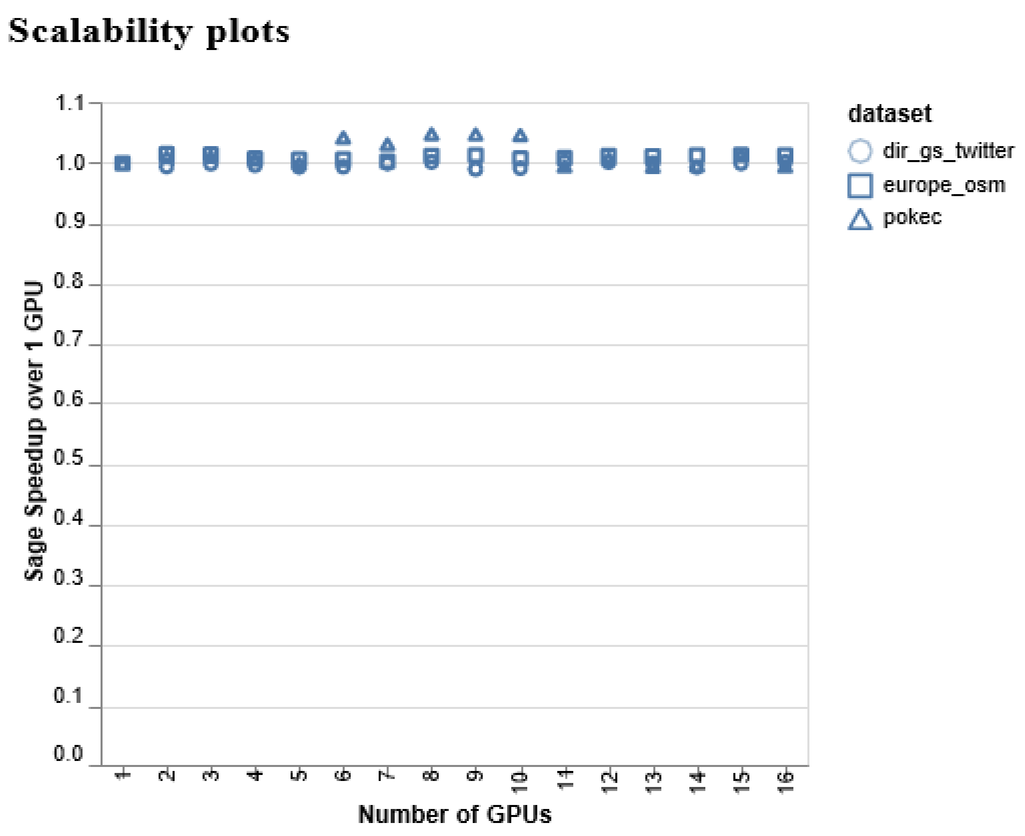

Figure 2: Sage: Speedup over 1 GPU vs. Number of GPUs

6 APPLICATION CLASSIFICATION

The **Phase 1 writeup** contains a detailed description of the application.

From the Phase 1 writeup:

> 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.

6.1 Scalability Summary

Bottlenecked by network bandwidth between GPUs

6.2 Summary of Results

We re-forumlate the **application_classification** workload to improve memory locality and admit a natural multi-GPU implementation. We then parallelized the core computational region of **application_classification** across GPUs. For the kernels in that region that do not require communication between GPUs, we attain near-perfect scaling. Runtime of the entire application remains bottlenecked by network bandwidth between GPUs. However, mitigating this bottleneck should be possible further optimization of the memory layout.

6.3 Summary of Implementation

The Phase 1 single-GPU implementation is **here**.

application_classification consists of two regions: - Region 1: initialization of distance and feature matrices - Region 2: iterative loop consisting of of message passing operations and matrix normalization operations

Region 2 accounts for the majority of runtime. For example, in our single-GPU implementation running on the **rmat18 application_classification** benchmark dataset, Region 1 takes 37ms (20% of runtime) and Region 2 takes 157ms (80% of runtime). As such, we focused on parallelizing Region 2 across GPUs. A multi-GPU implementation of Region 1 would also be possible, but with diminishing returns.

Upon examination of the Phase 1 **application_classification** implementation, we determined that most of the matrices could be transposed to attain better memory locality. In the original implementation, there were a number of column-wise operations (max reduce on columns; softmax normalization of columns). Transposing these matrices converts these into row-wise operations, and yields a substantial speedup. For example, on the **rmat18** benchmark dataset, this reformulation yields a 6.44x speedup on a single GPU.

"Transposing" the problem also makes it more suitable for multi-GPU parallelism, via row-wise chunking of the data matrices. Chunks are manually scattered across GPUs using **cudaMemcpy**. Most of the kernels in Region 2 require *no* communication between GPUs, which leads to good scaling. The small amount of communication that is required is done by enabling peer access, with remote memory loads / stores happening over NVLink.

Because it is not a canonical graph workload, **application_classification** is written outside of Gunrock using the **thrust** and **cub** libraries (as in HIVE Phase 1).

6.4 How to Run this Application on NVIDIA's DGX-2

6.4.1 Prerequisites

The setup process assumes **Anaconda** is already installed.

```
git clone \
    https://github.com/porumbes/application_classification \
    -b dev/mgpu_manual_reduce

cd application_classification

# prep binary input data
./hive-gen-data.sh

# build
make -j16
```

Verify git SHA: `commit 7e20dd05126c174c51b7155cb1f2f9e3084080b3`

6.4.2 Partitioning the Input Dataset

Partitioning is done automatically by the application.

6.4.3 Running the Application (Default Configurations)

```
./hive-mgpu-run.sh
```

This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application options as specified in `hive-ac-test.sh`. See **Running the Applications** for additional information.

6.4.3.1 Datasets

Default Locations:
`/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-2TB/application_classification/`
with subdirectory: `ac_JohnsHopkins_random`

Names:

```
rmat18
georgiyPattern
JohnsHopkins
```

6.4.4 Running the Application (Alternate Configurations)

6.4.4.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

6.4.4.2 hive-gen-data.sh

Unlike most of the other applications, Application Classification makes use of an additional script, **hive-gen-data.sh**, to generate necessary input. Please review the chapter on **Running the Applications** for information on running with additional datasets.

6.4.4.3 hive-ac-test.sh

Please see the Phase 1 single-GPU implementation details **here** for additional parameter information and review the provided script.

Given the setup in **hive-gen-data.sh**, modify the key-value store, **DATA_PATTERN** with the generated **rmat18_data.bin** as the key and the generated **georgiyPattern_pattern.bin** as the value. For example:
DATA_PATTERN["rmat18"]="georgiyPattern"

6.4.5 Output

No change from Phase 1.

6.5 Performance and Analysis

No change from Phase 1.

6.5.1 Implementation Limitations

Performance limitations regarding the size of the data matrices are mitigated by the multi-GPU approach – with this implementation, the maximum size of a problem instance should theoretically scale linearly with the number of GPUs. Practically, the current implementation still does Region 1 on a single GPU, which would create a bottleneck in terms of available memory.

Other performance limitations remain the same as in Phase 1.

6.5.2 Performance Limitations

From the perspective of a single GPU, there is no change from Phase 1.

From the perspective of the multi-GPU system, we are primarily bottlenecked by bandwidth across the NVLink network, which impacts both the runtime of the row scatter operation and the Region 2 kernels that require communication. This could be (partially) mitigated by additional optimizations – more details below.

6.6 Scalability Behavior

Scaling of the whole workload's runtime is not ideal, primarily because: a) because Region 1 is not parallelized across GPUs b) because scattering the rows of the matrices across GPUs takes time.

Region 1 would be relatively straightforward to distribute across GPUs. The runtime of the scatter could also be reduced via asynchronous memory copies (possibly launched from multiple CPU threads).

The scalability of Region 2 is limited by the couple of kernels that require communication between GPUs, which take ~5x longer to run w/ 4 GPUs than on a single GPU. Currently, we're bottlenecked by the bandwidth into GPU0 – scattering an additional datastructure across GPUs would reduce this load by a factor of **num_gpus**, and provide further speedup. However, this is slightly more complex than the current method, and has not yet been implemented.

6.7 Scalability Plots

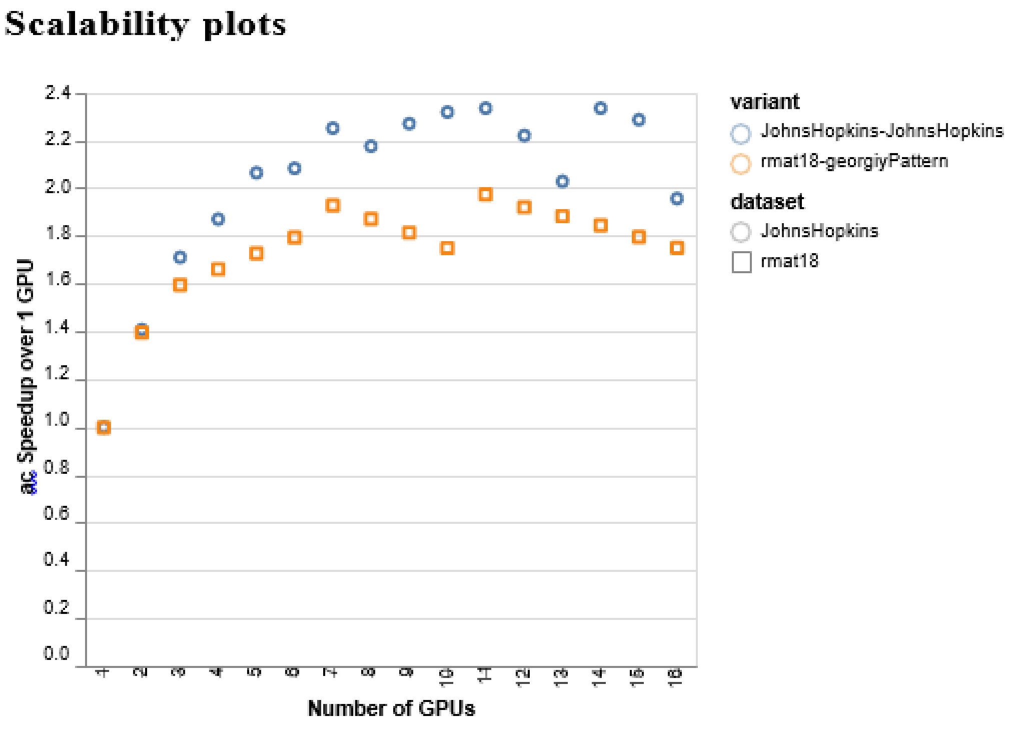

Figure 3: ac Speedup over 1 GPU vs. Number of GPUs

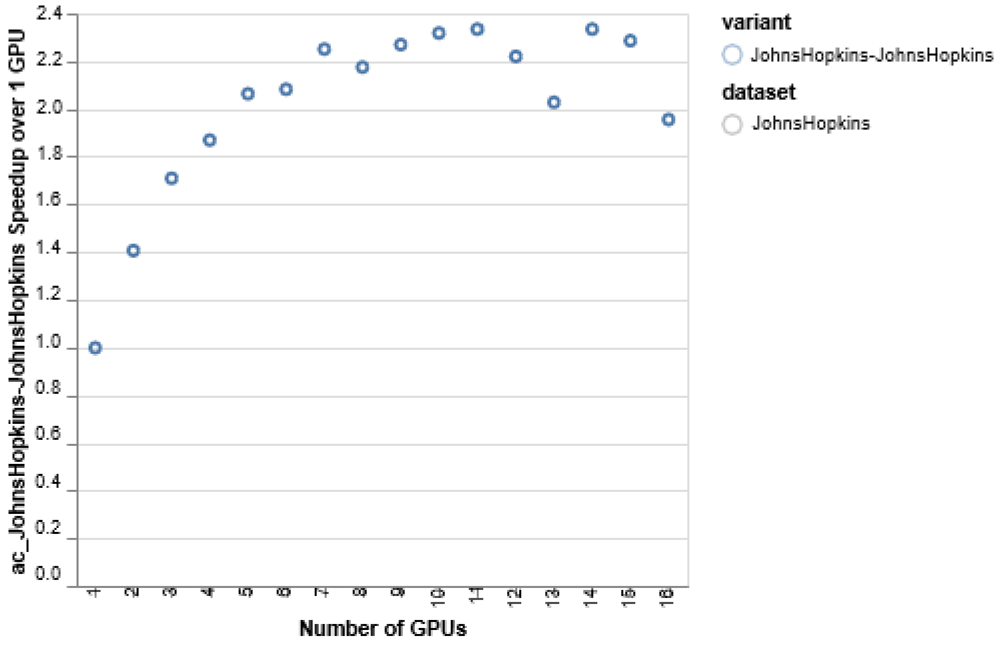

Figure 4: ac_JohnsHopkins-JohnsHopkins: Speedup over 1 GPU vs. Number of GPUs

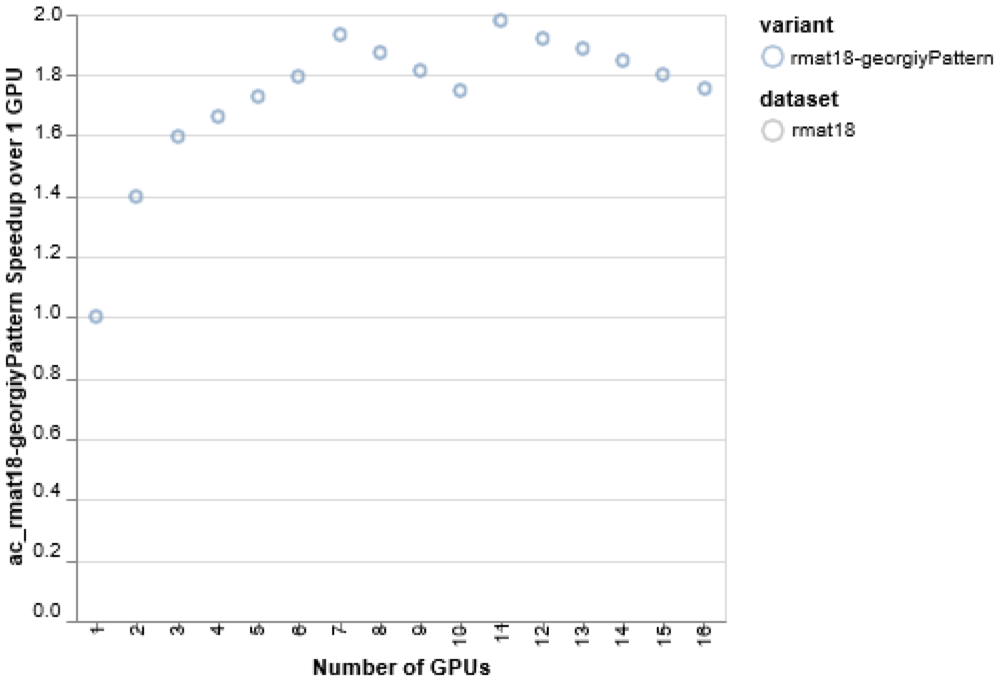

Figure 5: ac_rmat18-georgiyPattern: Speedup over 1 GPU vs. Number of GPUs

7 GEOLOCATION

From Phase 1 report:

> 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.

7.1 Scalability Summary

Bottlenecked by network bandwidth between GPUs

7.2 Summary of Results

We rely on Gunrock's multi-GPU `ForALL` operator to implement Geolocation as the entire behavior can be described within a single-loop like structure. The core computation focuses on calculating a spatial median, and for multi-GPU `ForAll`, that work is split such that each GPU gets an equal number of vertices to process. We see a minor speed-up on a DGX-A100 going from 1 to 3 GPUs on a twitter dataset, but in general, due to the communication over the GPU-GPU interconnects for all the neighbors of each vertex, there's a general pattern of slowdown going from 1 GPU to multiple GPUs, and no scaling is observed.

7.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.

We parallelize across GPUs by using multi-GPU **ForAll** operator that splits the latitude and longitude arrays of Geolocation algorithm equally over multiple devices. For more detail on how **ForAll** was written to be multi-GPU can be found in **Gunrock's ForAll Operator** section of the report. One optimization that we experimented with was using **BlockLoads** and shared memory (fast memory), to collectively load and process latitudes and longitudes in fast memory.

7.3.1 Differences in Implementation from Phase 1

No change from Phase 1.

7.4 How to Run this Application on NVIDIA's DGX-2

7.4.1 Prerequisites

```
git clone https://github.com/gunrock/gunrock -b mgpu-geo
mkdir build
cd build/
cmake ..
make -j16 geo
```

Verify git SHA: `commit b6e928b118f7ce792f82291cee5aa5d32547aaa3`

7.4.2 Partitioning the Input Dataset

Partitioning is handled automatically. Geolocation relies on Gunrock's multi-GPU `ForALL` operator and its frontier vertices are split evenly across all available GPUs (see **Gunrock's `ForAll` Operator** for more details).

7.4.3 Running the Application (Default Configurations)

From the `build` directory

```
cd ../examples/geo/
./hive-mgpu-run.sh
```

This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application option as specified in `hive-geo-test.sh`. Please see **Running the Applications** for more information.

7.4.3.1 Datasets

Default Locations:

```
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-
2TB/geolocation/twitter/graph
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-
2TB/geolocation/instagram/graph
```

Names:

```
twitter
instagram
```

7.4.4 Running the Application (Alternate configurations)

7.4.4.1 hive-mgpu-run.sh

modify `geo_iter` and `spatial_iter` to change the values of `--geo-iter` and `--spatial-iter`, respectively, passed to `hive-geo-test.sh`. please see the phase 1 single-gpu implementation details **here** for additional parameter information.

modify `output_dir` to store generated output and json files in an alternate location.

7.4.4.2 hive-geo-test.sh

please review the provided script and see the **running the applications** chapter for details on running with additional datasets.

7.4.5 Output

no change from phase 1.

7.4.6 Performance and Analysis

no change from phase 1.

7.4.7 Implementation Limitations

No change from Phase 1.

7.4.8 Performance limitations

Single-GPU: No change from Phase 1.
Multiple-GPUs: Performance bottleneck is the remote memory accesses from one GPU to another GPU's memory through NVLink. What we observed was if we simply extend `ForAll` from single to multiple GPUs, the remote memory accesses to neighbor's latitude and longitude arrays cause NVLink's network bandwidth to be the bottleneck for the entire application.

7.5 Scalability Behavior

Scaling is not ideal because we perform too many remote memory accesses causing the GPU to be constantly waiting to compute, therefore wasting the potential that GPU's throughput offers us. We require an efficient way to broadcast the latitudes and longitudes of a vertex to all other GPUs' local memory in between each iteration, which can help mitigate this issue and may result in better scaling characteristics. One possible way to achieve this in future work is by not using a `ForAll` and instead more specialized operators, designed with access patterns of these applications in mind (see **Gunrock's `ForAll` Operator** for more information).

7.6 Scalability Plots

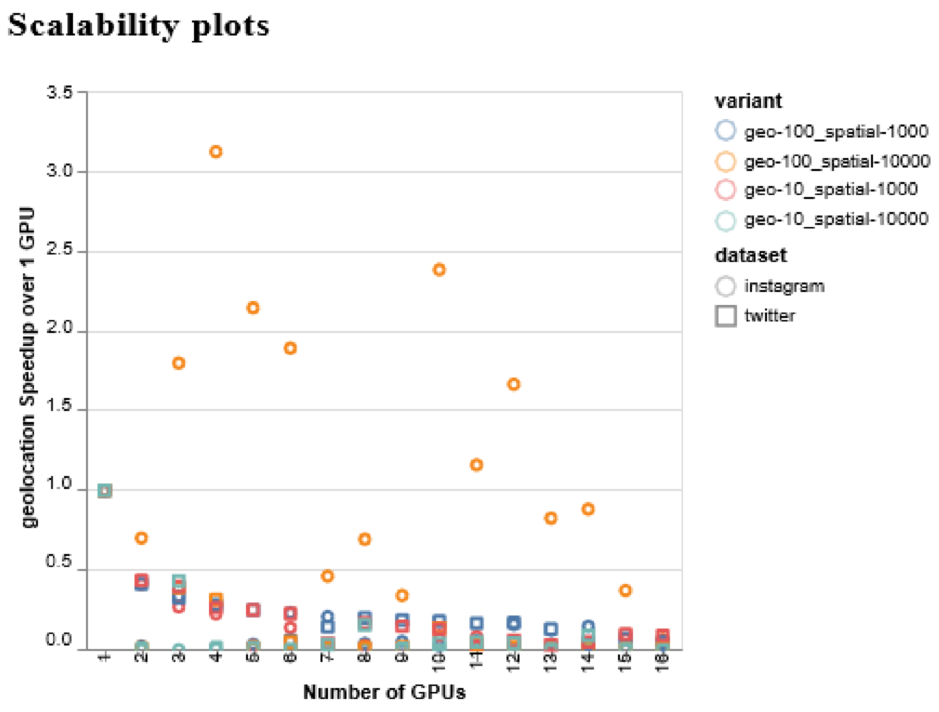

Figure 6: Geolocation: Speedup over 1 GPU vs. Number of GPUs

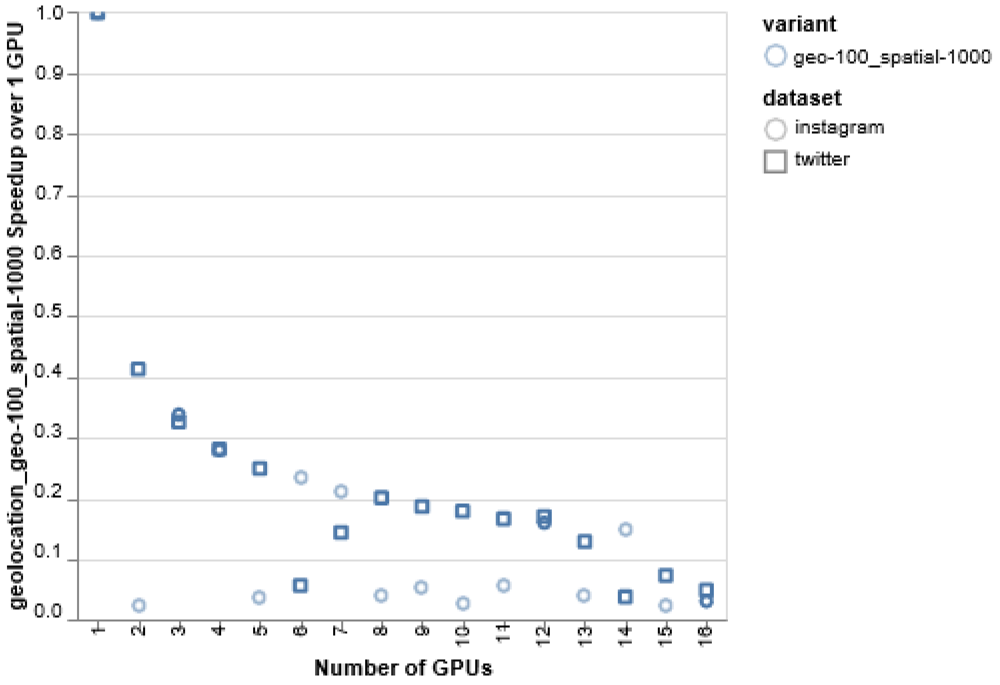

Figure 7: Geolocation_geo-100_spatial-10000: Speedup over 1 GPU vs. Number of GPUs

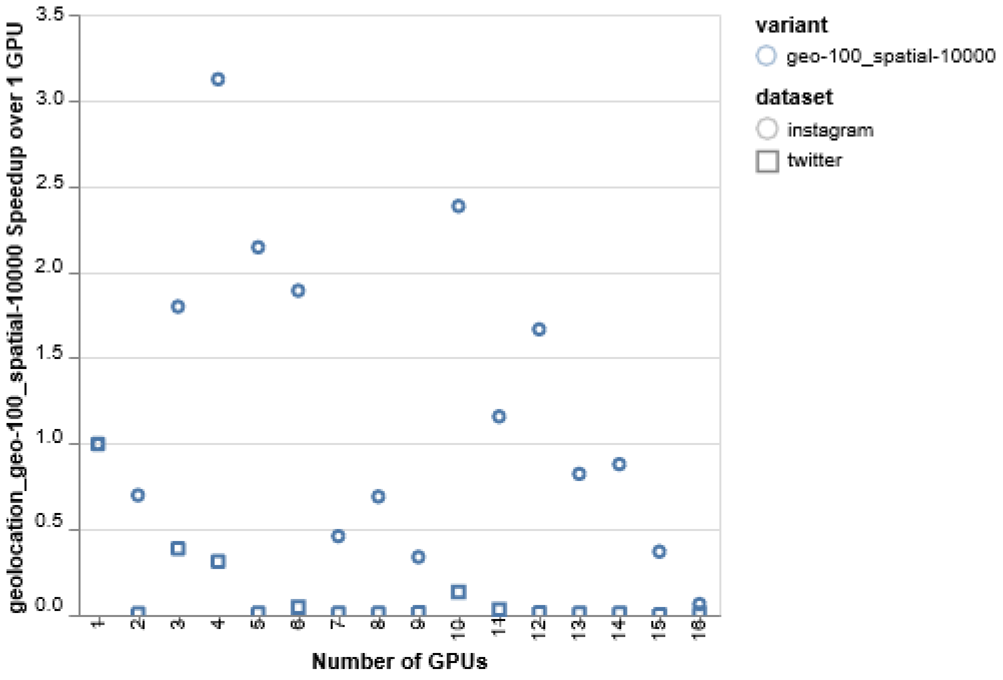

Figure 8: Geolocation_geo-100_spatial-10000: Speedup over 1 GPU vs. Number of GPUs

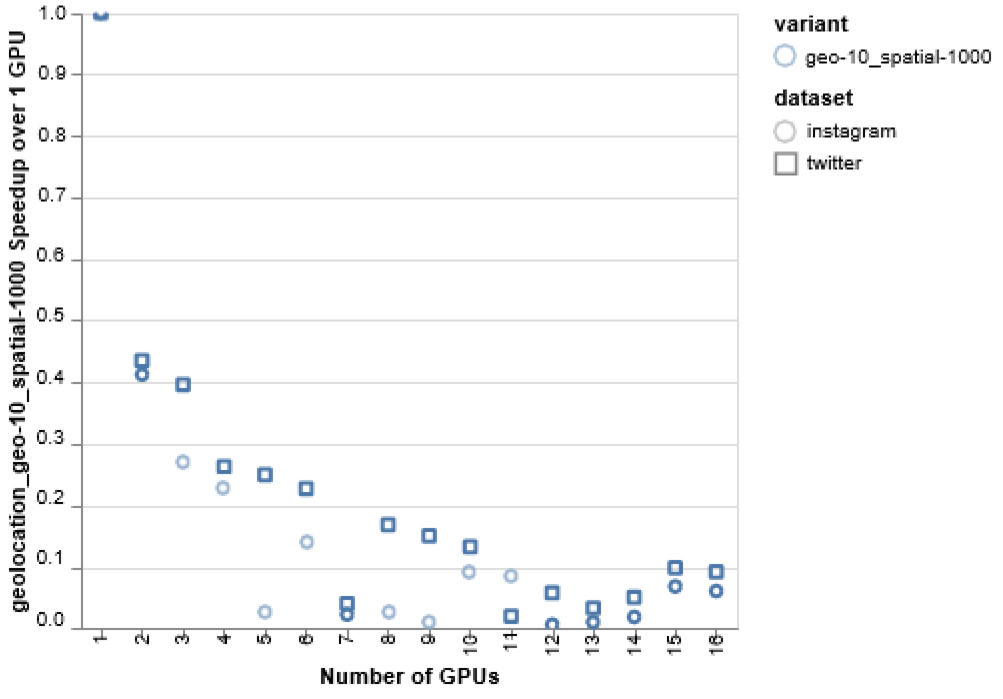

Figure 9: Geolocation_geo-10_spatial-1000: Speedup over 1 GPU vs. Number of GPUs

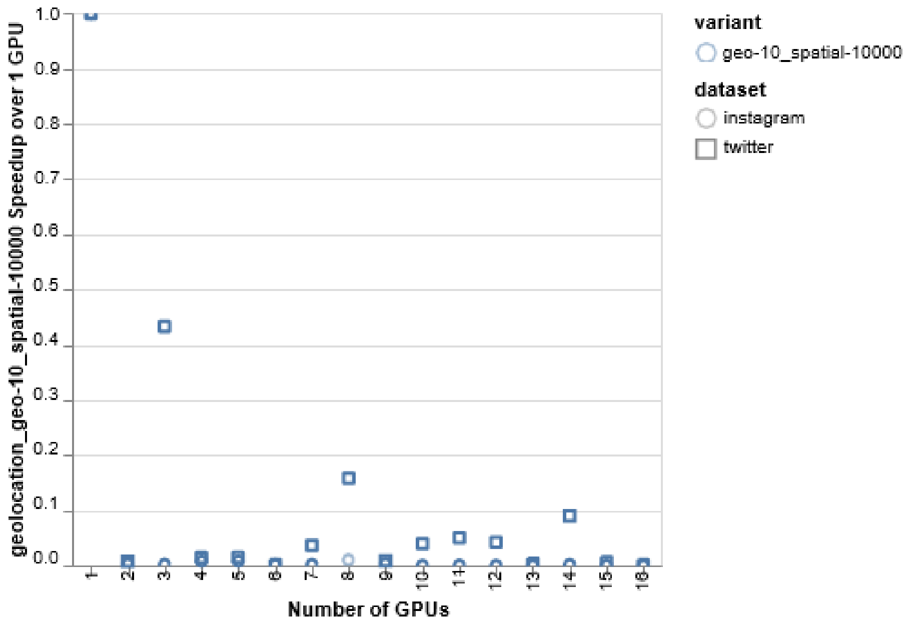

Figure 10: Geolocation_geo-10_spatial-10000: Speedup over 1 GPU vs. Number of GPUs

8 COMMUNITY DETECTION (LOUVAIN)

The **Phase 1 writeup** contains a detailed description of the application.

From the Phase 1 writeup:

> Community detection in graphs means grouping vertices together, so that those vertices that are closer (have more connections) to each other are placed in the same cluster. A commonly used algorithm for community detection is Louvain (**https://arxiv.org/pdf/0803.0476.pdf**).

8.1 Scalability Summary

Application is nonfunctional

8.2 Summary of Results

The application has a segmentation fault and is currently nonfunctional.

8.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.

We parallelize across GPUs by utilizing Gunrock's multi-GPU `ForAll` operator described **here**.

8.3.1 Differences in Implementation from Phase 1

No change from Phase 1.

8.4 How to Run This Application on NVIDIA's DGX-2

8.4.1 Prerequisites

```
git clone  https://github.com/gunrock/gunrock -b multigpu
mkdir build
cd build/
cmake ..
make -j16 louvain
```
Verify git SHA: `commit d70a73c5167c5b59481d8ab07c98b376e77466cc`

8.4.2 Partitioning the Input Dataset

Partitioning is handled automatically as Community Detection relies on Gunrock's multi-GPU `ForALL` operator and its data is split evenly across all available GPUs

8.4.3 Running the Application

Once functional, the application will follow the two script approach described in **Running the Applications** (i.e., using `hive-mgpu-run.sh` and `hive-louvain-test.sh` scripts).

8.4.3.1 Datasets

Final datasets will be listed when the application is functional.

8.4.4 Output

No change from Phase 1.

8.5 Performance and Analysis

No change from Phase 1.

8.5.1 Implementation Limitations

Currently nonfunctional.

8.5.2 Performance Limitations

Currently nonfunctional.

8.6 Scalability Behavior

Currently unavailable, but unlikely to scale given its `ForAll` based implementation. See **Gunrock's `ForAll` Operator** for additional information.

9 LOCAL GRAPH CLUSTERING (LGC)

The **Phase 1 writeup** contains a detailed description of the application.

From the Phase 1 writeup:

> 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).

9.1 Scalability Summary

Bottlenecked by single-GPU and communication

9.2 Summary of Results

We rely on Gunrock's multi-GPU `ForALL` operator to implement Local Graph Clustering and observe no scaling as we increase from one to sixteen GPUs. The application is likely bottlenecked by single-GPU filter and advance operators and communication across NVLink necessary to access arrays distributed across GPUs.

9.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.

We parallelize Local Graph Clustering by utilizing a multi-GPU `ForAll` operator that splits necessary arrays evenly across multiple GPUs. Additional information on multi-GPU `ForAll` can be found in **Gunrock's `ForAll` Operator** section of the report. In addition, this application depends on single-GPU implementations of Gunrock's advance and filter operations.

9.3.1 Differences in Implementation from Phase 1

No change from Phase 1.

9.4 How to Run this Application on NVIDIA's DGX-2

9.4.1 Prerequisites

```
git clone  https://github.com/gunrock/gunrock -b multigpu
mkdir build
cd build/
cmake ..
make -j16 pr_nibble
```

Verify git SHA: `commit 3e7d4f29f0222e9fd1f4e768269b704d6ebcd02c`

9.4.2 Partitioning the Input Dataset

Partitioning is handled automatically as Local Graph Clustering relies on Gunrock's multi-GPU `ForALL` operator and its frontier vertices are split evenly across all available GPUs. Please refer to the chapter on **Gunrock's `ForAll` Operator** for additional information.

9.4.3 Running the Application (Default Configurations)

From the `build` directory
```
cd ../examples/pr_nibble/
./hive-mgpu-run.sh
```
This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application option as specified in `hive-pr_nibble-test.sh`. See **Running the Applications** for additional information.

9.4.3.1 Datasets

Default Locations:
`/home/u00u7u37rw7AjJoA4e357/data/gunrock/gunrock_dataset/mario-2TB/large`
Names:
```
hollywood-2009
europe_osm
```

9.4.4 Running the Application (Alternate Configurations)

9.4.4.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

9.4.4.2 hive-geo-test.sh

Modify `APP_OPTIONS` to specify alternate `--src` and `--max-iter` values. Please see the Phase 1 single-GPU implementation details **here** for additional parameter information.

Please review the provided script and see **Running the Applications** for details on running with additional datasets.

9.4.5 Output

No change from Phase 1.

9.5 Performance and Analysis

No change from Phase 1.

9.5.1 Implementation Limitations

No change from Phase 1.

9.5.2 Performance Limitations

Single-GPU: No change from Phase 1.
Multiple-GPUs: The performance bottleneck is likely due to single-GPU implementations of advance and filter operations randomly accessing numerous arrays distributed across multiple GPUs.

9.6 Scalability Behavior

We observe no scaling with Local Graph Clustering as currently implemented. Please see the chapter on **Gunrock's `ForAll` Operator** for a discussion on future directions around more specialized operators to be designed with communication patterns in mind.

9.7 Scalability Plots

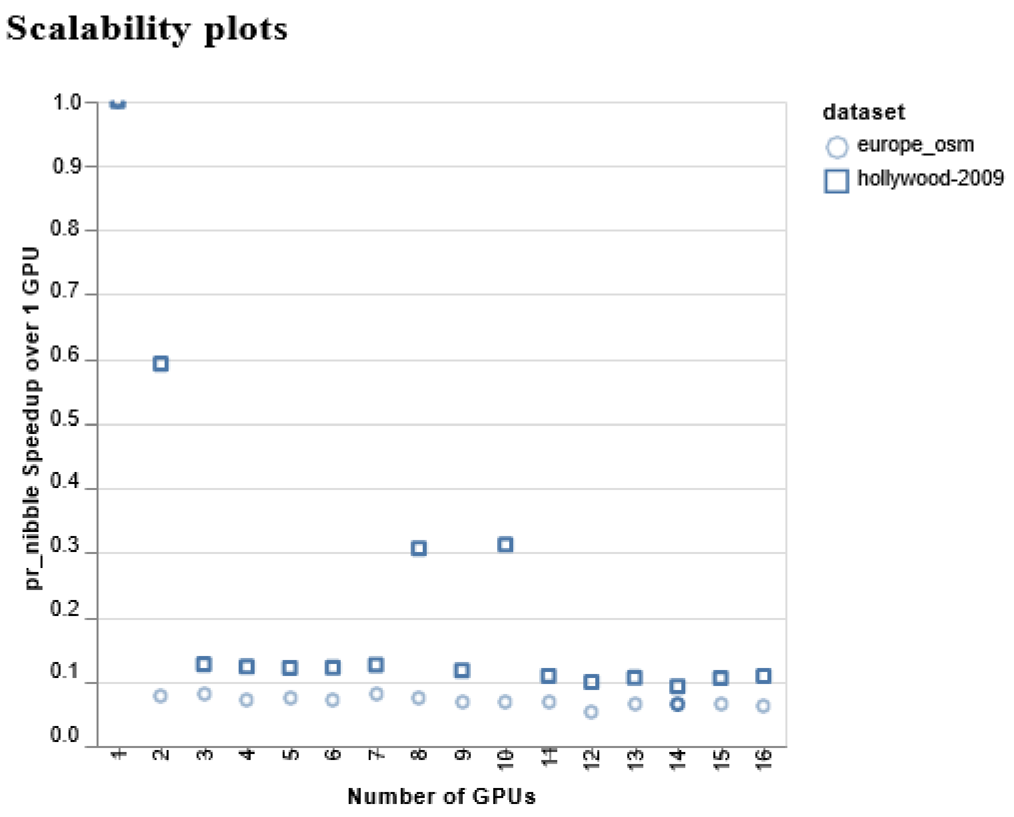

Figure 11: pr_nibble: Speedup over 1 GPU vs. Number of GPUs

10 GRAPH PROJECTIONS

The **Phase 1 writeup** contains a detailed description of the application.

From the Phase 1 writeup:

> 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.

Note that mathematically this reduces to a sparse-sparse matrix multiplication of `G`'s adjacency matrix.

10.1 Scalability Summary

Limited by load imbalance

10.2 Summary of Results

We implemented a multi-GPU version of sparse-sparse matrix multiplication, based on chunking the rows of the left hand matrix. This yields a communication-free implementation with good scaling properties. However, our current implementation remains partially limited by load imbalance across GPUs.

10.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.
In Phase 1, we had two implementations: one using GraphBLAS and one using Gunrock. The GraphBLAS implementation is more obviously distributed across GPUs, so we build off of that implementation.

`graph_projections` for a symmetric graph is mathematically `H = A @ A`, where `A` is the adjacency matrix of graph `G`. One way to easily parallelize this operation across GPUs is by partitioning on the rows of the left hand matrix:

```
H = row_stack([A[start_row:end_row] @ A for start_row, end_row in
partition(n_rows)])
```

We parallelize across GPUs by copying the adjacency matrix of `G` to each GPU. Then, for each GPU, we determine the chunk of rows of the left hand matrix that will be computed on, and each GPU computes `A[start_row:end_row] @ A` for its respective chunk. No communication between GPUs is required, except for the initial scatter.The adjacency matrix `A` is assumed to be randomly permuted and the number of rows in a chunk is constant. This leads to a coarse-grained load balancing – each chunk has *roughly* the same number of nonzero entries. However, some rows in a power law graph may have orders of magnitude more non-zero entries than others, which does lead to some load imbalance in this application.

10.3.1 Differences in Implementation from Phase 1

The multi-GPU implementation consists of wrapper code around the Phase 1 implementation that distributes `A` across all of the GPUs, launches independent computation on each GPU, and collects the results.

10.4 How to Run This Application on NVIDIA's DGX-2

10.4.1 Prerequisites

```
git clone https://github.com/owensgroup/graphblas_proj -b dev/mgpu2
cd graphblas_proj
make -j16
```
Verify git SHA: `commit c55074593fac49de088ca9afa9d2e82422bccda4`

10.4.2 Partitioning the Input Dataset

Data partitioning occurs at runtime whereby matrix `A` is distributed across all available GPUs. Please see the summary above for more information.

10.4.3 Running the Application (Default Configurations)

```
./hive-mgpu-run.sh
```
This will launch jobs that sweep across 1 to 16 GPU configurations per dataset as specified in `hive-proj-test.sh`. See **Running the Applications** for additional information.

10.4.3.1 Datasets

Default Locations:
```
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-
2TB/proj_movielens
```
Names:
```
ml_1000000
ml_5000000
ml_full
```

10.4.4 Running the Application (Alternate Configurations)

10.4.4.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

10.4.4.2 hive-proj-test.sh

Please review the provided script and see **Running the Applications** for details on running with additional datasets. In addition matrix market `.mtx` must first be converted to binary as follows:

```
# convert data to binary
python data/mtx2bin.py --inpath data/ml_full.mtx
```

10.4.5 Output

No change from Phase 1.

10.5 Performance and Analysis

No change from Phase 1.

10.5.1 Implementation Limitations

Implementation limitations are largely the same as in Phase 1.

The input graph still must fit onto a single GPU, as this parallelization strategy requires the adjacency matrix `A` to be replicated across all GPUs.

However, in the multi-GPU implementation, only `1 / num_gpus` of the *output* adjacency matrix `H` must fit on a GPU. This is important, because `H` tends to be a dense matrix, which causes us to run out of GPU memory for even medium-sized graphs `G`. Thus, the multi-GPU implementation does allow us to run `graph_projections` on larger graphs, approximately linearly with the number of GPUs used.

10.5.2 Performance Limitations

No change from Phase 1 – in the multi-GPU setting, each GPU is doing almost exactly the same operations as the single-GPU setting, albeit on a subset of the left hand matrix rows.

10.6 Scalability Behavior

Scaling is predominantly limited by the presence of load imbalance due to the constant size chunking of rows. To attain perfect scaling, we would want to use a dynamically allocated chunk of the left hand matrix such that the number of nonzero elements is approximately equal, rather than such that the number of *rows* is approximately equal. This is a somewhat non-trivial optimization – we'd need either some heuristic for creating chunks of rows with *approximately* the same number of nonzero elements *or* we'd need to add support for accumulating values across GPUs. However, we do expect that one of these approaches would lead to further improvements in scaling.

The time it takes to copy the input adjacency matrix `A` to each GPU also contributes to some imperfect scaling, though the cost of this operation tends to be small compared to the cost of the actal computation.

10.7 Scalability Plots

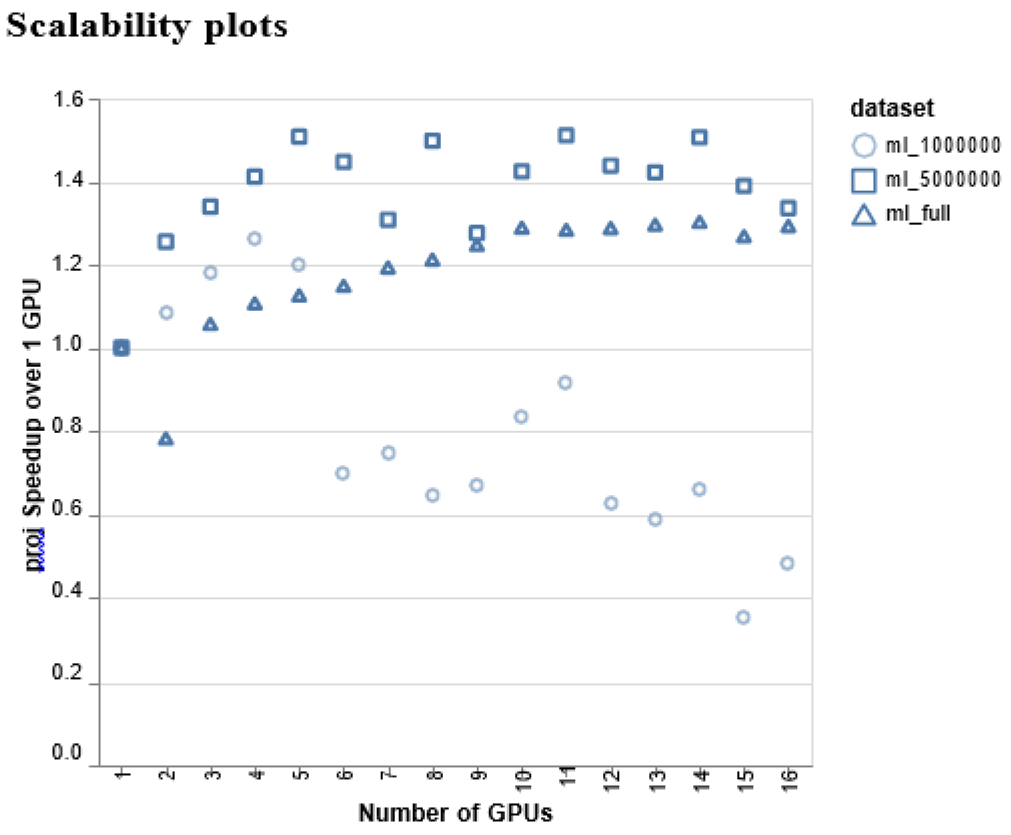

Figure 12: proj: Speedup over 1 GPU vs. Number of GPUs

11 GRAPHSEARCH

The Phase 1 report for GraphSearch can be found **here**.

The graph search (GS) workflow is 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.

11.1 Scalability Summary

Bottlenecked by network bandwidth between GPUs

11.2 Summary of Results

We rely on a Gunrock’s multi-GPU `ForALL` operator to implement GraphSearch as the entire behavior can be described within a single-loop like structure. The core computation focuses on determining which neighbor to visit next based on uniform, greedy, or stochastic functions. Each GPU is given an equal number of vertices to process. No scaling is observed, and in general we see a pattern of decreased performance as we move from 1 to 16 GPUs due to random neighbor access across GPU interconnects.

11.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.
We parallelize across GPUs by using a multi-GPU `ForAll` operator that splits arrays equally across GPUs. For more detail on how `ForAll` was written to be multi-GPU can be found in **Gunrock’s `ForAll` Operator** section of the report.

11.3.1 Differences in implementation from Phase 1

No change from Phase 1.

11.4 How to Run This Application on NVIDIA’s DGX-2

11.4.1 Prerequisites

```
git clone  https://github.com/gunrock/gunrock -b multigpu
mkdir build
cd build/
cmake ..
make -j16 rw
```
Verify git SHA: `commit d70a73c5167c5b59481d8ab07c98b376e77466cc`

11.4.2 Partitioning the Input Dataset

How did you do this? Command line if appropriate.
include a transcript

11.4.3 Running the Application (Default Configurations)

From the `build` directory

```
cd ../examples/rw/
./hive-mgpu-run.sh
```

This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application option as specified across three different test scripts:

- `hive-rw-undirected-uniform.sh`
- `hive-rw-directed-uniform.sh`
- `hive-rw-directed-greedy.sh`

Please see **Running the Applications** for additional information.

11.4.3.1 Datasets

Default Locations:
`/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-2TB/graphsearch`
Names:
`dir_gs_twitter`
`gs_twitter.values`

11.4.4 Running the Application (Alternate Configurations)

11.4.4.1 hive-mgpu-run.sh

Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

11.4.4.2 Additional hive-rw-*.sh scripts

This application relies on Gunrock's random walk `rw` primitive. Modify `WALK_MODE` to control the application's `--walk-mode` parameter and specify `--undirected` as `true` or `false`. Please see the Phase 1 single-GPU implementation details **here** for additional parameter information.

11.4.5 Output

No change from Phase 1.

11.5 Performance and Analysis

No change from Phase 1.

11.5.1 Implementation Limitations

No change from Phase 1.

11.5.2 Performance Limitations

Single-GPU: No change from Phase 1.
Multiple-GPUs: Performance bottleneck is the remote memory accesses from one GPU to another GPU's memory through NVLink.

11.6 Scalability Behavior

GraphSearch scales poorly due to low compute (not enough computation per memory access) and high communication costs due to random access patterns (across multiple GPUs) characteristic to the underlying "random walk" algorithm used.

11.7 Scalability Plots

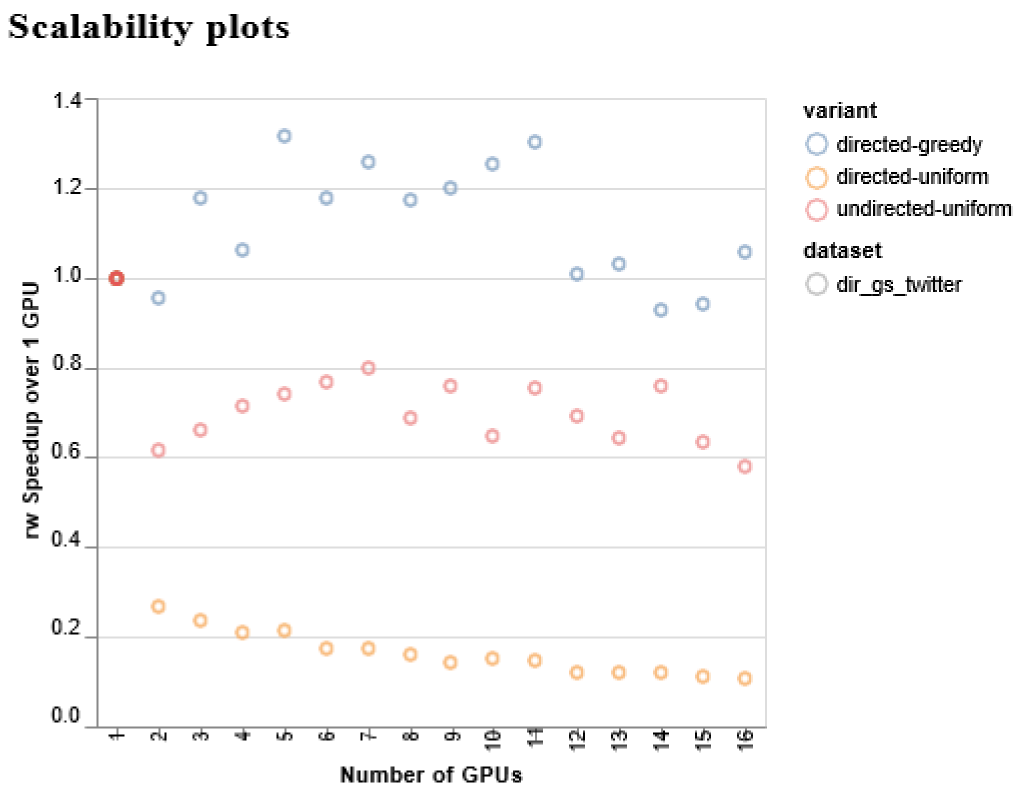

Figure 13: rw: Speedup over 1 GPU vs. Number of GPUs

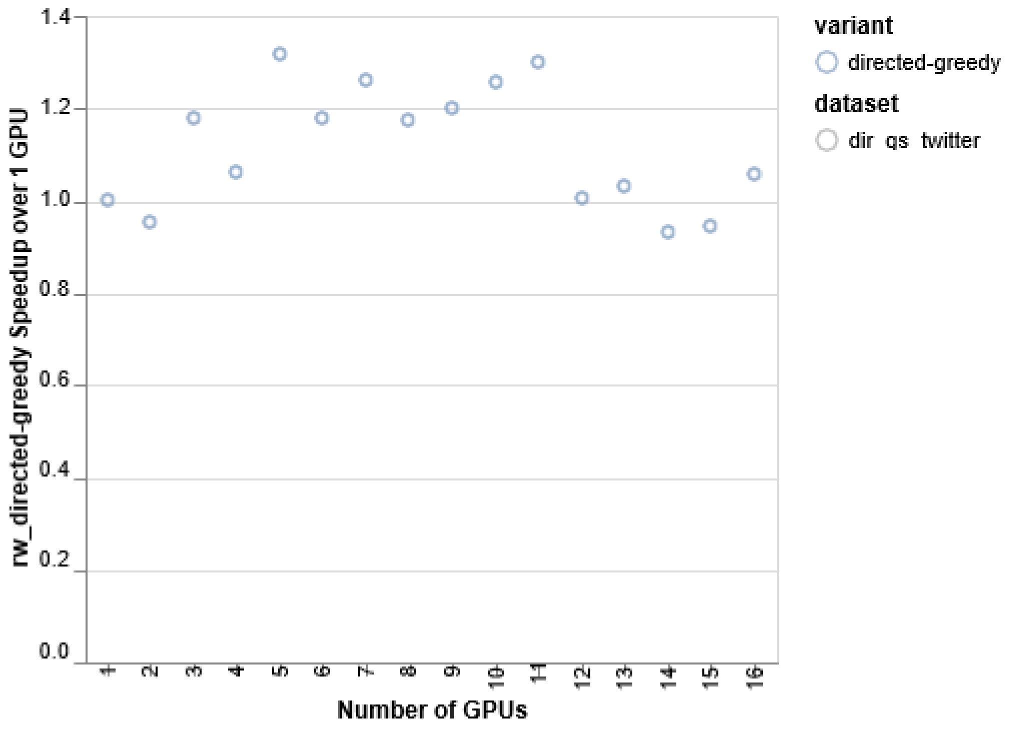

Figure 14: rw_directed-greedy: Speedup over 1 GPU vs. Number of GPUs

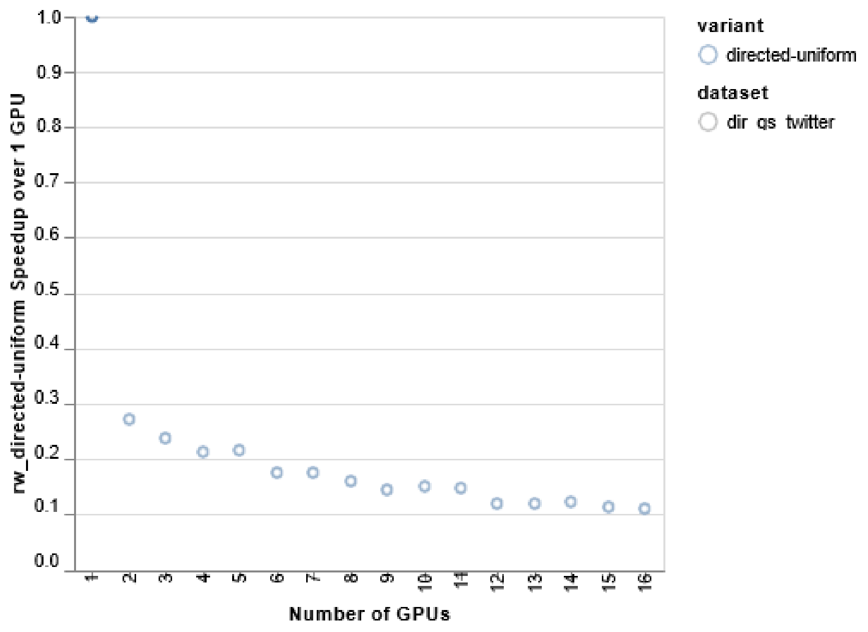

Figure 15: rw_directed-uniform: Speedup over 1 GPU vs. Number of GPUs

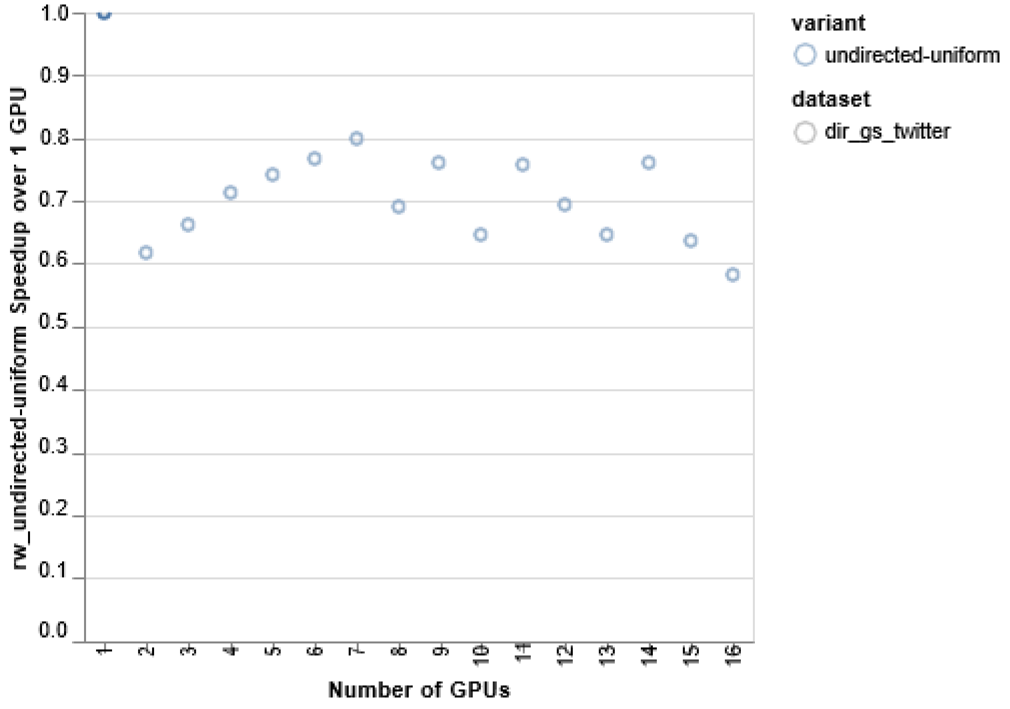

Figure 16: rw_undirected-uniform: Speedup over 1 GPU vs. Number of GPUs

12 SEEDED GRAPH MATCHING (SGM)

The Phase 1 report for SGM can be found **here**.
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.

12.1 Scalability Summary

We observe great scaling

12.2 Summary of Results

Multi-GPU SGM experiences considerable speed-ups over single GPU implementation with a near linear scaling if the dataset being processed is large enough to fill up the GPU. We notice that ~1 million nonzeros sparse-matrix is a decent enough size for us to show decent scaling as we increase the number of GPUs. The misalignment for this implementation is also synthetically generated (just like it was for Phase 1, the bottleneck is still the `|V|x|V|` allocation size).

12.3 Summary of Gunrock Implementation

The Phase 1 single-GPU implementation is **here**.
We parallelize across GPUs by scaling the per-iteration linear assignment problem. In our multi-GPU implementation we ignore the preprocessing step of sparse general matrix multiplication of given input matrices and the trace of matrix products at the very end. For the assignment problem, we use the auction algorithm (also described in the Phase 1 report), where each CUDA block gets a row of the cost matrix and does parallel reductions across the entries of the row using all available threads (with the help of NVIDIA's CUB library). This allows us to map our rows to each block and explore parallelism within a single row of the matrix in a single-GPU, and split the number of rows across multiple GPUs. Our auction algorithm is implemented using a 2-step process (2-kernels with one fill operation to reset the maximum bids):

1. **Bidding:** Each bidder chooses an object which brings him/her the best value (benefit-price).
2. **Assign:** Each object chooses a bidder which has the highest bid, and assigns itself to him/her as well as increases the object's price.

Our experiments conclude that this "bidding" step was the bottleneck for our auction algorithm, and is the only kernel needed to be parallelized across multiple GPUs. For our assignment kernel, it was more effective to use one block to do the final assignment and use one volatile variable to compute the convergence metric.

12.3.1 Differences in Implementation from Phase 1

We now assign each row of the matrix to an entire block instead of a CUDA thread, and process the row in parallel instead of sequentially.

12.4 How to Run this Application on NVIDIA's DGX-2

12.4.1 Prerequisites

```
git clone https://github.com/owensgroup/SGM -b mgpu
cd SGM/test/
make
```
Verify git SHA: `commit d41a43d5653455c1adc59841499ce84a63ecd2db`

12.4.2 Partitioning the Input Dataset

Data partitioning occurs at runtime whereby matrix rows are split across multiple GPUs. Please see the summary above for more information.

12.4.3 Running the Application (Default Configurations)

From the `test` directory
```
./hive-mgpu-run.sh
```
This will launch jobs that sweep across 1 to 16 GPU configurations per dataset as specified in `hive-sgm-test.sh`. **(see `hive_run_apps_phase2.md` for more info)**.
Please note: due to an intermittent bug (occassional infinite loop) in the implementation, the scheduled SLURM job is set to timeout after three minutes (all used datasets should complete in under one minute).

12.4.3.1 Datasets

Default Locations:
```
/home/u00u7u37rw7AjJoA4e357/data/gunrock/hive_datasets/mario-2TB/seeded-graph-matching/connectome
```
Names:
```
DS00833
DS01216
DS01876
DS03231
DS06481
DS16784
```

12.4.4 Running the Application (Alternate Configurations)

12.4.4.1 hive-mgpu-run.sh

Due to the bug mentioned above, a user may wish to increase or decrease the SLURM job cancellation time. Modify the **`--time`** options shown here:

```
SLURM_CMD="srun --cpus-per-gpu 2 -G $i -p $PARTITION_NAME -N 1 --time=3:00 "
```

Modify **`OUTPUT_DIR`** to store generated output and json files in an alternate location.

12.4.4.2 hive-sgm-test.sh

A tolerance value can be specified by setting a value in **`APP_OPTIONS`**
Please review the provided script and see **Running the Applications** for information on running with additional datasets.

12.4.5 Output

No change from Phase 1.

12.5 Performance and Analysis

No change from Phase 1.

12.5.1 Implementation Limitations

No change from Phase 1.

12.5.2 Performance Limitations

Single-GPU: No change from Phase 1.
Multiple-GPUs: Our multi-GPU implementation does not consider the SpGEMM preprocessing step. As SpGEMM is one of the core computations for many other algorithms, one future opportunity will be to scale a load-balanced SpGEMM to a multi-GPU system using merge-based decomposition. CUDA's new virtual memory APIs also allow us to map and unmap physical memory chunks to a contiguous virtual memory array, which can be used to perform and store SpGEMM in its sparse-format without relying on an intermediate dense representation and a conversion to sparse output.

12.6 Scalability Behavior

We observe great scaling for our bidding kernel as we increase the number of GPUs. If the input matrix is large enough, the rows can be easily split across multiple GPUs, and each GPU processes its equal share of rows, where within a GPU, each CUDA block processes one complete row.

12.7 Scalability Plots

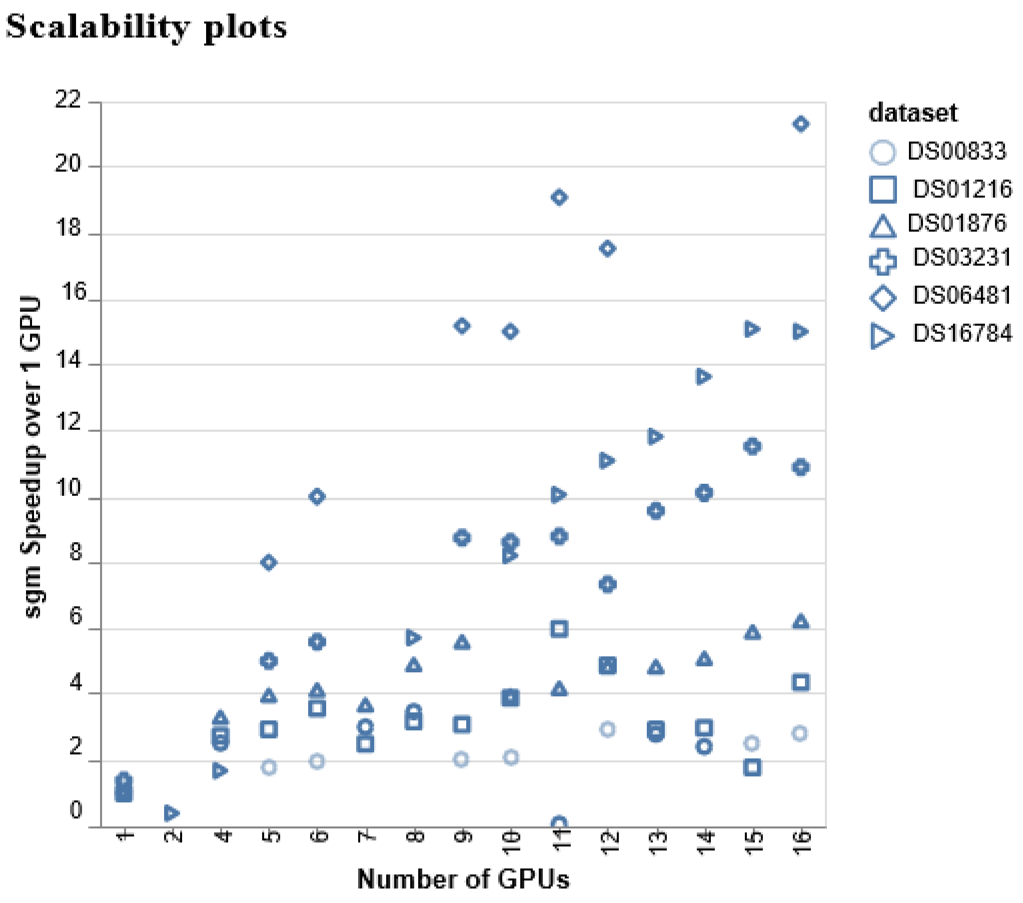

Figure 17: sgm: Speedup over 1 GPU vs. Number of GPUs

13 SPARSE FUSED LASSO

The Phase 1 report for SFL is found **here**.

Given a graph where each vertex on the graph has a weight, *sparse fused lasso (SFL)*, also named *sparse graph trend filter (GTF)*, tries to learn a new weight for each vertex that is (1) sparse (most vertices have weight 0), (2) close to the original weight in the l2 norm, and (3) close to its neighbors' weight(s) in the l1 norm. This algorithm is usually used in main trend filtering (denoising). For example, an image (grid graph) with noisy pixels can be filtered with this algorithm to get a new image without the noisy pixels, which are "smoothed out" by its neighbors. **https://arxiv.org/abs/1410.7690**

13.1 Scalability Summary

Maxflow kernel is serial

13.2 Summary of Results

Sparse Fused Lasso (or Sparse Graph Trend Filtering) relies on a Maxflow algorithm. As highlighted in the Phase 1 report, a sequential implementation of Maxflow outperforms a single-GPU implementation, and the actual significant core operation of SFL is a serial normalization step that cannot be parallelized to a single GPU, let alone multiple GPUs. Therefore, we refer readers to the phase 1 report for this workload. Parallelizing across multiple GPUs is not beneficial.

14 VERTEX NOMINATION

The **Phase 1 writeup** contains a detailed description of the application. The most important point to note is that **`vertex_nomination`** is a "multiple-source shortest paths" algorithm. The algorithm description and implementation are identical to canonical single-source shortest paths (SSSP), with the minor modification that the search starts from multiple vertices instead of one.

14.1 Scalability Summary

We observe weak scaling

14.2 Summary of Results

We implemented **`vertex_nomination`** as a standalone CUDA program, and achieve good weak scaling performance by eliminating communication during the **`advance`** phase of the algorithm and using a frontier representation that allows an easy-to-compute reduction across devices.

14.3 Summary of Gunrock Implementation and Differences from Phase 1

The Phase 1 single-GPU implementation is **here**.
In Phase 1, **`vertex_nomination`** was implemented for a single GPU using the Gunrock framework. However, The Phase 2 multi-GPU implementation required some functionality that is not currently available in Gunrock, so we implemented it as a standalone CUDA program (using the **`thrust`** and **`NCCL`** libraries).

Specifically, the multi-GPU **`vertex_nomination`** uses a fixed-size (boolean or integer) array to represent the input and output frontiers, while Gunrock predominantly uses a dynamically-sized list of vertex IDs. The fixed-size representation admits a more natural multi-GPU implementation, and avoids a complex merge / deduplication step in favor of a cheap **`or`** reduce step.

As described in the Phase 1 report, the core kernel in **`vertex_nomination`** is the following advance:

```
def _advance_op(src, dst, distances):
    src_distance   = distances[src]
    edge_weight    = edge_weights[(src, dst)]
    new_distance   = src_distance + edge_weight
    old_distance   = distances[dst]
    distances[dst] = min(old_distance, new_distance)
    return new_distance < old_distance
```

which runs in a loop like the following pseudocode:

```
# advance
thread_parallel for src in input_frontier:
  thread_parallel for dst in src.neighbors():
    if _advance_op(src, dst, distances):
      output_frontier.add(dst)
```

In the multi-GPU implementation, the loop instead looks like the following pseudocode:

```
# advance per GPU
device_parallel for device in devices:
  thread_parallel for src in local_input_frontiers[device].get_chunk(device):
    thread_parallel for dst in src.neighbors():
      if _advance_op(src, dst, local_distances[device]):
        local_output_frontiers[device][dst] = True

# reduce across GPUs
local_input_frontiers = all_reduce(local_output_frontiers, op="or")
device_parallel for device in devices:
  local_output_frontiers[device][:] = False

local_distances = all_reduce(local_distances, op="min")
```

In the per-GPU **advance** phase, each device has

- a *local* replica of the complete input graph
- a chunk of nodes it is responsible for computing on
- a *local* copy of the **input_frontier** that is read from
- a *local* copy of the **output_frontier** that is written to
- a *local* copy of the **distance** array that is read / written

This data layout means that no communication between devices is required during the advance phase.

During the **reduce** phase,

- the local output frontiers are reduced with the **or** operator (remember they are boolean masks)
- the local **distances** arrays are reduced with the **min** operator

After this phase, the copies of the input frontiers and the computed distances are the same on each device. In our implementation, these reduces uses the **ncclAllReduce** function from NVIDIA's **nccl** library.

14.4 How to Run this Application on NVIDIA's DGX-2

14.4.1 Prerequisites

The setup process assumes Anaconda is already installed.

```
git clone  git clone https://github.com/porumbes/mgpu_sssp -b main
cd mgpu_sssp
bash install.sh # downloads and compiles NVIDIA's nccl library
make
```

Verify git SHA: `commit 4f93307e7a0aa7f71e8ab024771e950e40247a4e`

14.4.2 Partitioning the Input Dataset

The input graph is replicated across all devices.

Each device is reponsible for running the `advance` operation on a subset of nodes in the graph (eg, `GPU:0` operates on node range `[0, n_nodes / n_gpus]`, `GPU:1` on `[n_nodes / n_gpus + 1, 2 * n_nodes / n_gpus]`, etc). Assuming a random node labeling, this correspond to a random partition of nodes across devices.

14.4.3 Running the Application (Default Configurations)

```
./hive-mgpu-run.sh
```

This will launch jobs that sweep across 1 to 16 GPU configurations per dataset and application options as specified in **`hive-vn-test.sh` (see `hive_run_apps_phase2.md` for more info)**.

14.4.3.1 Datasets

Default Locations:
`/home/u00u7u37rw7AjJoA4e357/data/gunrock/gunrock_dataset/mario-2TB/large`
Names:

```
chesapeake
rmat18
rmat20
rmat22
rmat24
enron
hollywood-2009
indochina-2004
```

14.4.4 Running the Application (Alternate Configurations)

14.4.4.1 hive-mgpu-run.sh

Modify **NUM_SEEDS** to specify the number of seed locations to be used by **hive-vn-test.sh**. Modify **OUTPUT_DIR** to store generated output and json files in an alternate location.

14.4.4.2 hive-vn-test.sh

Please see the Phase 1 single-GPU implementation details **here** for additional parameter information, review the provided script, and see **Running the Applications** chapter for details on running with additional datasets.

14.4.5 Output

No change from Phase 1.

14.5 Performance and Analysis

No change from Phase 1.

14.5.1 Implementation Limitations

Implementation limitations are largely the same as in the Phase 1 Gunrock-based implementation.

Note that in the current implementation, the *entire* input graph is replicated across all devices. That means that this implementation cannot run on datasets that are large than the memory of a single GPU.

14.5.2 Performance Limitations

The **advance** phase does not include any communication between devices, so the performance limitations are the same as in Phase 1.

The **reduce** phase requires copying and reducing **local_output_frontiers** and **local_distances** across GPUs, and is memory bandwidth bound.

14.6 Scalability Behavior

Scaling is not perfectly ideal because of the time taken by the **reduce** phase, which is additional work in the multi-GPU setting that is not present in the single-GPU case. As the number of GPUs increases, the cost of this communication increases relative to the per-GPU cost of computation, which limits weak scaling of our implementation.

Scaling is primarily limited by the current restriction that the entire input graph must fit in a single GPU's memory. From a programming perspective, it would be straightforward to partition the input graph across GPUs; however, this would lead to remote memory accesses in the **advance** phase and impact performance substantially.

14.7 Scalability Plots

Scalability plots

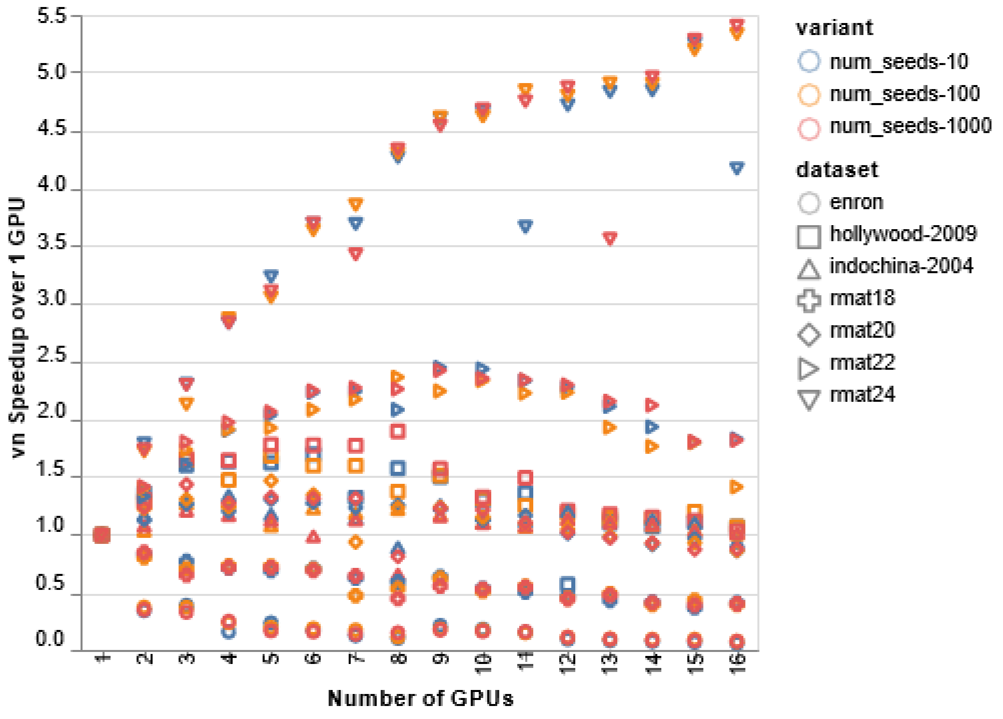

Figure 18: vn: Speedup over 1 GPU vs. Number of GPUs

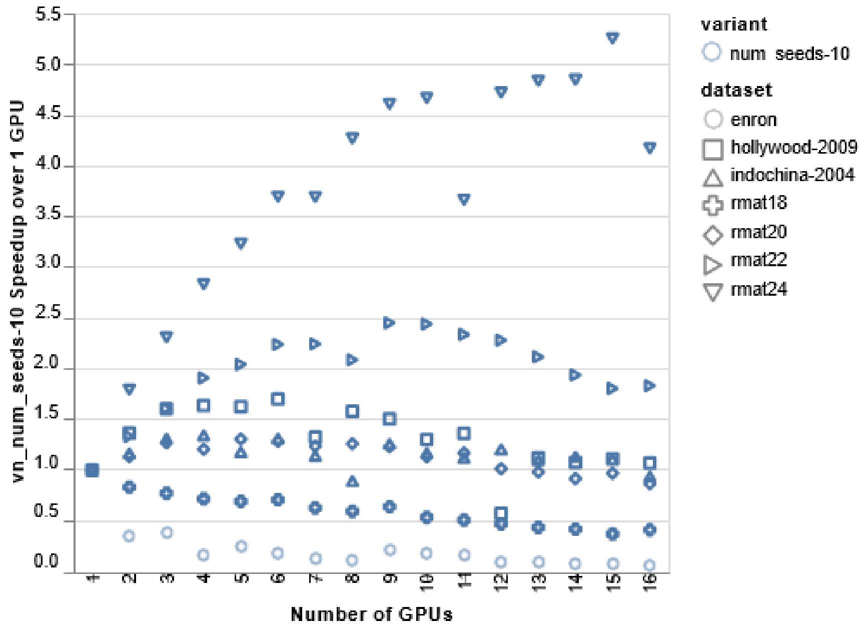

Figure 19: vn_num_seeds-100: speedup over 1 GPU vs. number of GPUs

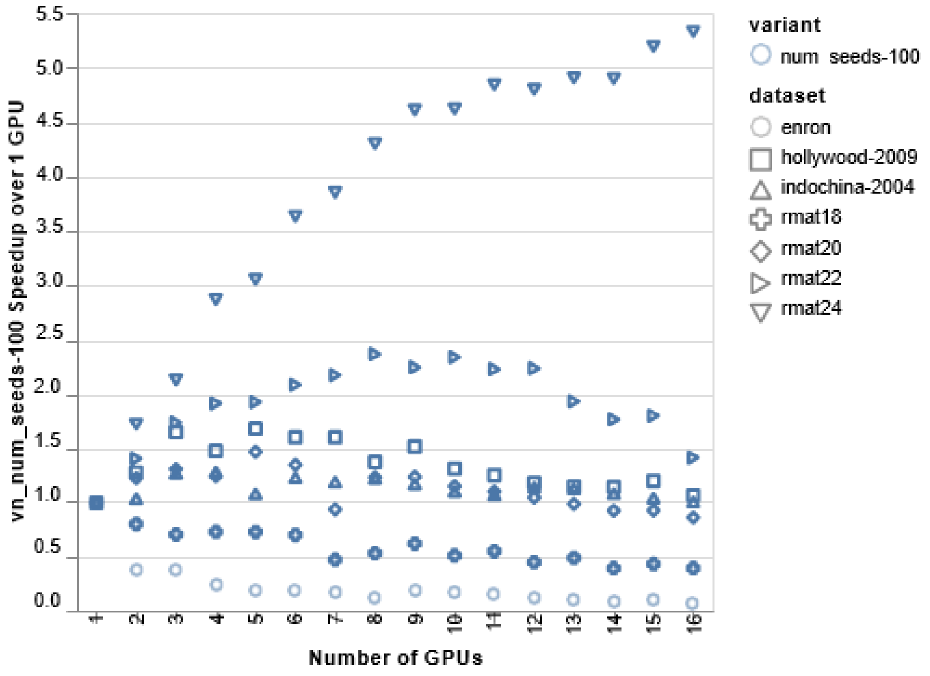

Figure 20: vn_num_seeds-100: Speedup over 1 GPU vs. Number of GPUs

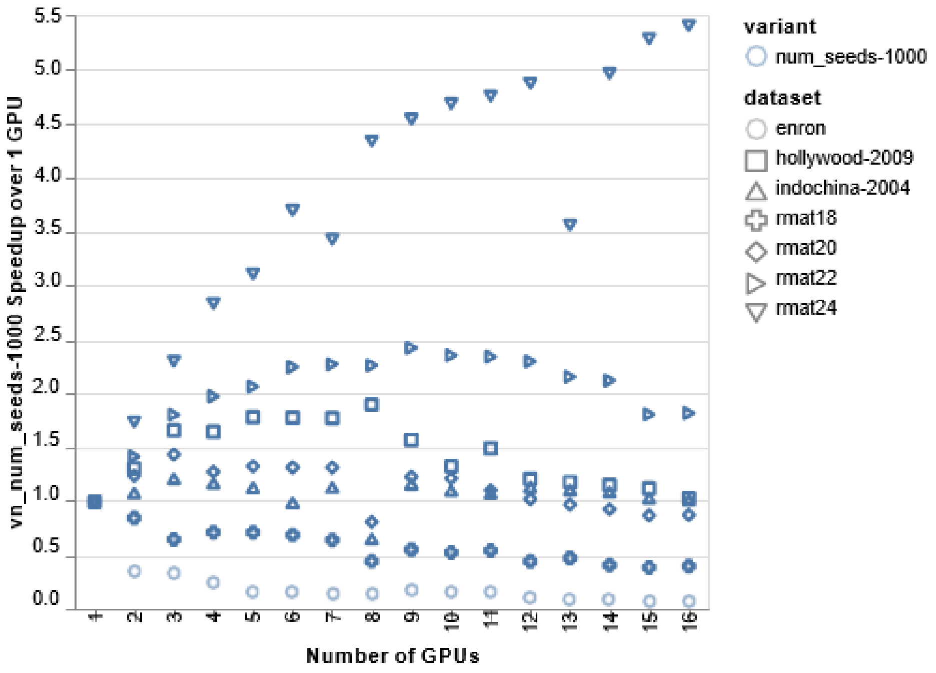

Figure 21: vn_num_seeds-1000: Speedup over 1 GPU vs. Number of GPUs

15 TABLES OF PERFORMANCE RESULTS

Table 1. Tabular Data for SS

primitive	dataset	num-gpus	avg-process-time	speedup
SS	pokec	1	89.0786	1
SS	pokec	2	95.0498	0.937178
SS	pokec	3	89.979	0.989993
SS	pokec	4	98.9693	0.900062
SS	pokec	5	91.8863	0.969443
SS	pokec	6	94.2538	0.945092
SS	pokec	7	90.5149	0.984132
SS	pokec	8	95.51	0.932662
SS	pokec	9	94.2913	0.944717
SS	pokec	10	98.4475	0.904833
SS	pokec	11	91.6058	0.972412
SS	pokec	12	99.5731	0.894605
SS	pokec	13	95.3088	0.934631
SS	pokec	14	101.498	0.877637
SS	pokec	15	95.4893	0.932864
SS	pokec	16	101.367	0.878776

Table 2. Tabular Data for Sage

primitive	dataset	num-gpus	avg-process-time	speedup
Sage	dir_gs_twitter	1	3836.83	1
Sage	dir_gs_twitter	2	3855.94	0.995042
Sage	dir_gs_twitter	3	3839.15	0.999395
Sage	dir_gs_twitter	4	3848.06	0.99708
Sage	dir_gs_twitter	5	3859.28	0.994182
Sage	dir_gs_twitter	6	3857.84	0.994554
Sage	dir_gs_twitter	7	3836.05	1.0002
Sage	dir_gs_twitter	8	3826.94	1.00258
Sage	dir_gs_twitter	9	3873.4	0.990559
Sage	dir_gs_twitter	10	3867.13	0.992163
Sage	dir_gs_twitter	11	3826.04	1.00282
Sage	dir_gs_twitter	12	3828.73	1.00212
Sage	dir_gs_twitter	13	3834.75	1.00054
Sage	dir_gs_twitter	14	3860.03	0.993988
Sage	dir_gs_twitter	15	3835.66	1.00031

Sage	dir_gs_twitter	16	3826.97	1.00258
Sage	europe_osm	1	26701.5	1
Sage	europe_osm	2	26293.9	1.0155
Sage	europe_osm	3	26311.5	1.01482
Sage	europe_osm	4	26490.1	1.00798
Sage	europe_osm	5	26534.2	1.00631
Sage	europe_osm	6	26525.4	1.00664
Sage	europe_osm	7	26611.8	1.00337
Sage	europe_osm	8	26362.8	1.01285
Sage	europe_osm	9	26350.5	1.01332
Sage	europe_osm	10	26486.8	1.00811
Sage	europe_osm	11	26464.3	1.00896
Sage	europe_osm	12	26365.3	1.01275
Sage	europe_osm	13	26385.3	1.01199
Sage	europe_osm	14	26352.5	1.01324
Sage	europe_osm	15	26331.8	1.01404
Sage	europe_osm	16	26342.7	1.01362
Sage	pokec	1	922.788	1
Sage	pokec	2	913.131	1.01058
Sage	pokec	3	912.807	1.01093
Sage	pokec	4	916.144	1.00725
Sage	pokec	5	922.868	0.999913
Sage	pokec	6	884.999	1.0427
Sage	pokec	7	894.332	1.03182
Sage	pokec	8	879.899	1.04874
Sage	pokec	9	880.266	1.04831
Sage	pokec	10	881.411	1.04694
Sage	pokec	11	927.09	0.995359
Sage	pokec	12	916.011	1.0074
Sage	pokec	13	927.485	0.994936
Sage	pokec	14	926.408	0.996093
Sage	pokec	15	909.469	1.01464
Sage	pokec	16	927.374	0.995055

Table 3. Tabular Data for ac_JohnsHopkins-JohnsHopkins

primitive	dataset	variant	num-gpus	avg-process-time	speedup
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	1	56.249	1
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	2	39.965	1.40746
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	3	32.89	1.71022
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	4	30.078	1.8701
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	5	27.249	2.06426
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	6	27.002	2.08314
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	7	24.983	2.25149
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	8	25.85	2.17598
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	9	24.776	2.2703
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	10	24.261	2.31849
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	11	24.091	2.33486
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	12	25.329	2.22074
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	13	27.731	2.02838
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	14	24.089	2.33505
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	15	24.602	2.28636
ac	JohnsHopkins	JohnsHopkins-JohnsHopkins	16	28.751	1.95642

Table 4. Tabular Data for ac_rmat18-georgiy Pattern

primitive	dataset	variant	num-gpus	avg-process-time	speedup
ac	rmat18	rmat18-georgiyPattern	1	159.124	1
ac	rmat18	rmat18-georgiyPattern	2	114.038	1.39536
ac	rmat18	rmat18-georgiyPattern	3	99.739	1.5954
ac	rmat18	rmat18-georgiyPattern	4	95.815	1.66074
ac	rmat18	rmat18-georgiyPattern	5	92.112	1.72751
ac	rmat18	rmat18-georgiyPattern	6	88.729	1.79337
ac	rmat18	rmat18-georgiyPattern	7	82.535	1.92796
ac	rmat18	rmat18-georgiyPattern	8	85.006	1.87191
ac	rmat18	rmat18-georgiyPattern	9	87.712	1.81416
ac	rmat18	rmat18-georgiyPattern	10	90.967	1.74925
ac	rmat18	rmat18-georgiyPattern	11	80.588	1.97454
ac	rmat18	rmat18-georgiyPattern	12	82.836	1.92095
ac	rmat18	rmat18-georgiyPattern	13	84.485	1.88346
ac	rmat18	rmat18-georgiyPattern	14	86.28	1.84427
ac	rmat18	rmat18-georgiyPattern	15	88.604	1.7959
ac	rmat18	rmat18-georgiyPattern	16	90.913	1.75029

Table 5. Tabular Data for Geolocation_geo-100_spatial-1000

primitive	dataset	variant	num-gpus	avg-process-time	speedup
geolocation	instagram	geo-100_spatial-1000	1	365.553	1
geolocation	instagram	geo-100_spatial-1000	2	913.354	0.400231
geolocation	instagram	geo-100_spatial-1000	3	5465.63	0.0668821
geolocation	instagram	geo-100_spatial-1000	4	7779.85	0.0469871
geolocation	instagram	geo-100_spatial-1000	5	1553.92	0.235245
geolocation	instagram	geo-100_spatial-1000	6	9279.76	0.0393925
geolocation	instagram	geo-100_spatial-1000	7	5977.18	0.061158
geolocation	instagram	geo-100_spatial-1000	8	5177.91	0.0705986
geolocation	instagram	geo-100_spatial-1000	9	2073.7	0.176281
geolocation	instagram	geo-100_spatial-1000	10	9613.43	0.0380252
geolocation	instagram	geo-100_spatial-1000	11	9236.87	0.0395754
geolocation	instagram	geo-100_spatial-1000	12	9813.92	0.0372484
geolocation	instagram	geo-100_spatial-1000	13	7871.26	0.0464414
geolocation	instagram	geo-100_spatial-1000	14	2473.71	0.147775
geolocation	instagram	geo-100_spatial-1000	15	10716.2	0.0341122
geolocation	instagram	geo-100_spatial-1000	16	5316.04	0.0687641
geolocation	twitter	geo-100_spatial-1000	1	825.405	1
geolocation	twitter	geo-100_spatial-1000	2	1991.51	0.414461
geolocation	twitter	geo-100_spatial-1000	3	7929.66	0.104091
geolocation	twitter	geo-100_spatial-1000	4	2965.83	0.278305
geolocation	twitter	geo-100_spatial-1000	5	18663.4	0.0442258
geolocation	twitter	geo-100_spatial-1000	6	3571.15	0.231131
geolocation	twitter	geo-100_spatial-1000	7	3778.44	0.218451
geolocation	twitter	geo-100_spatial-1000	8	3967.11	0.208062
geolocation	twitter	geo-100_spatial-1000	9	4348.86	0.189798
geolocation	twitter	geo-100_spatial-1000	10	37281.6	0.0221397
geolocation	twitter	geo-100_spatial-1000	11	18382.1	0.0449027
geolocation	twitter	geo-100_spatial-1000	12	5153.53	0.160163
geolocation	twitter	geo-100_spatial-1000	13	5075.25	0.162633
geolocation	twitter	geo-100_spatial-1000	14	5553.3	0.148633
geolocation	twitter	geo-100_spatial-1000	15	5557.06	0.148533
geolocation	twitter	geo-100_spatial-1000	16	25369.9	0.0325348

Table 6. Tabular data for geolocation_geo-100_spatial-10000

primitive	dataset	variant	num-gpus	avg-process-time	speedup
geolocation	instagram	geo-100_spatial-10000	1	142295	1
geolocation	instagram	geo-100_spatial-10000	2	69469.4	2.04832
geolocation	instagram	geo-100_spatial-10000	3	57560	2.47213
geolocation	instagram	geo-100_spatial-10000	4	121071	1.1753
geolocation	instagram	geo-100_spatial-10000	5	87918.5	1.61849
geolocation	instagram	geo-100_spatial-10000	6	286668	0.496378
geolocation	instagram	geo-100_spatial-10000	7	546899	0.260186
geolocation	instagram	geo-100_spatial-10000	8	70707.8	2.01244
geolocation	instagram	geo-100_spatial-10000	9	205886	0.691137
geolocation	instagram	geo-100_spatial-10000	10	508681	0.279734
geolocation	instagram	geo-100_spatial-10000	11	214711	0.662729
geolocation	instagram	geo-100_spatial-10000	12	138831	1.02495
geolocation	instagram	geo-100_spatial-10000	13	189341	0.751532
geolocation	instagram	geo-100_spatial-10000	14	354348	0.40157
geolocation	instagram	geo-100_spatial-10000	15	406091	0.350403
geolocation	instagram	geo-100_spatial-10000	16	307838	0.462242
geolocation	twitter	geo-100_spatial-10000	1	3052.13	1
geolocation	twitter	geo-100_spatial-10000	2	7042.78	0.43337
geolocation	twitter	geo-100_spatial-10000	3	173864	0.0175547
geolocation	twitter	geo-100_spatial-10000	4	10163.9	0.300291
geolocation	twitter	geo-100_spatial-10000	5	240802	0.0126749
geolocation	twitter	geo-100_spatial-10000	6	127611	0.0239175
geolocation	twitter	geo-100_spatial-10000	7	206707	0.0147655
geolocation	twitter	geo-100_spatial-10000	8	162298	0.0188057
geolocation	twitter	geo-100_spatial-10000	9	78975.6	0.0386465
geolocation	twitter	geo-100_spatial-10000	10	154822	0.0197138
geolocation	twitter	geo-100_spatial-10000	11	359897	0.00848056
geolocation	twitter	geo-100_spatial-10000	12	58212	0.0524312
geolocation	twitter	geo-100_spatial-10000	13	187845	0.0162481
geolocation	twitter	geo-100_spatial-10000	14	48034.6	0.0635402
geolocation	twitter	geo-100_spatial-10000	15	216210	0.0141165
geolocation	twitter	geo-100_spatial-10000	16	356902	0.00855171

Table 7. Tabular Data for Geolocation_geo-10_spatial-1000

primitive	dataset	variant	num-gpus	avg-process-time	speedup
geolocation	instagram	geo-10_spatial-1000	1	109.508	1
geolocation	instagram	geo-10_spatial-1000	2	275.592	0.397356
geolocation	instagram	geo-10_spatial-1000	3	366.617	0.298699
geolocation	instagram	geo-10_spatial-1000	4	505.492	0.216637
geolocation	instagram	geo-10_spatial-1000	5	620.655	0.176439
geolocation	instagram	geo-10_spatial-1000	6	3502.39	0.0312667
geolocation	instagram	geo-10_spatial-1000	7	776.309	0.141062
geolocation	instagram	geo-10_spatial-1000	8	6200.84	0.0176602
geolocation	instagram	geo-10_spatial-1000	9	6442.11	0.0169988
geolocation	instagram	geo-10_spatial-1000	10	9923.67	0.011035
geolocation	instagram	geo-10_spatial-1000	11	1308.17	0.0837109
geolocation	instagram	geo-10_spatial-1000	12	7834.48	0.0139777
geolocation	instagram	geo-10_spatial-1000	13	9601.38	0.0114054
geolocation	instagram	geo-10_spatial-1000	14	2753.98	0.0397636
geolocation	instagram	geo-10_spatial-1000	15	5054.8	0.0216642
geolocation	instagram	geo-10_spatial-1000	16	5243.03	0.0208864
geolocation	twitter	geo-10_spatial-1000	1	309.224	1
geolocation	twitter	geo-10_spatial-1000	2	656.088	0.471315
geolocation	twitter	geo-10_spatial-1000	3	910.158	0.339748
geolocation	twitter	geo-10_spatial-1000	4	1222.04	0.253039
geolocation	twitter	geo-10_spatial-1000	5	1434.19	0.215609
geolocation	twitter	geo-10_spatial-1000	6	1460.74	0.211691
geolocation	twitter	geo-10_spatial-1000	7	1773.42	0.174366
geolocation	twitter	geo-10_spatial-1000	8	14617.6	0.0211542
geolocation	twitter	geo-10_spatial-1000	9	2169.88	0.142508
geolocation	twitter	geo-10_spatial-1000	10	2454.57	0.125979
geolocation	twitter	geo-10_spatial-1000	11	2702.54	0.11442
geolocation	twitter	geo-10_spatial-1000	12	2707.44	0.114213
geolocation	twitter	geo-10_spatial-1000	13	21745.1	0.0142204
geolocation	twitter	geo-10_spatial-1000	14	11884.9	0.0260182
geolocation	twitter	geo-10_spatial-1000	15	3457.58	0.0894337
geolocation	twitter	geo-10_spatial-1000	16	3736.6	0.0827554

Table 8. Tabular Data for Geolocation_geo-10_spatial-10000

primitive	dataset	variant	num-gpus	avg-process-time	speedup
geolocation	instagram	geo-10_spatial-10000	1	108109	1
geolocation	instagram	geo-10_spatial-10000	2	42378.2	2.55104
geolocation	instagram	geo-10_spatial-10000	3	1662.53	65.0265
geolocation	instagram	geo-10_spatial-10000	4	108227	0.998907
geolocation	instagram	geo-10_spatial-10000	5	74429.4	1.4525
geolocation	instagram	geo-10_spatial-10000	6	137900	0.783967
geolocation	instagram	geo-10_spatial-10000	7	65735.9	1.64459
geolocation	instagram	geo-10_spatial-10000	8	90253.5	1.19783
geolocation	instagram	geo-10_spatial-10000	9	346777	0.311753
geolocation	instagram	geo-10_spatial-10000	10	86573.6	1.24875
geolocation	instagram	geo-10_spatial-10000	11	144650	0.747381
geolocation	instagram	geo-10_spatial-10000	12	466175	0.231906
geolocation	instagram	geo-10_spatial-10000	13	335880	0.321867
geolocation	instagram	geo-10_spatial-10000	14	394338	0.274152
geolocation	instagram	geo-10_spatial-10000	15	73649.9	1.46787
geolocation	instagram	geo-10_spatial-10000	16	113847	0.949599
geolocation	twitter	geo-10_spatial-10000	1	460029	1
geolocation	twitter	geo-10_spatial-10000	2	74893.2	6.14246
geolocation	twitter	geo-10_spatial-10000	3	6388.86	72.0049
geolocation	twitter	geo-10_spatial-10000	4	46411.6	9.91194
geolocation	twitter	geo-10_spatial-10000	5	67375.5	6.82784
geolocation	twitter	geo-10_spatial-10000	6	435663	1.05593
geolocation	twitter	geo-10_spatial-10000	7	264635	1.73835
geolocation	twitter	geo-10_spatial-10000	8	164129	2.80284
geolocation	twitter	geo-10_spatial-10000	9	291643	1.57737
geolocation	twitter	geo-10_spatial-10000	10	322431	1.42675
geolocation	twitter	geo-10_spatial-10000	11	54207.2	8.48649
geolocation	twitter	geo-10_spatial-10000	12	113762	4.04378
geolocation	twitter	geo-10_spatial-10000	13	332051	1.38542
geolocation	twitter	geo-10_spatial-10000	14	284508	1.61693
geolocation	twitter	geo-10_spatial-10000	15	119153	3.86084
geolocation	twitter	geo-10_spatial-10000	16	201496	2.28306

Table 9. Tabular Data for pr_nibble

primitive	dataset	num-gpus	avg-process-time	speedup
pr_nibble	europe_osm	1	1.06192	1
pr_nibble	europe_osm	2	13.71	0.0774555
pr_nibble	europe_osm	3	13.1001	0.0810614
pr_nibble	europe_osm	4	14.9431	0.0710639
pr_nibble	europe_osm	5	14.07	0.0754736
pr_nibble	europe_osm	6	14.9491	0.0710355
pr_nibble	europe_osm	7	13.272	0.0800115
pr_nibble	europe_osm	8	14.168	0.0749516
pr_nibble	europe_osm	9	15.3902	0.0689997
pr_nibble	europe_osm	10	15.671	0.0677631
pr_nibble	europe_osm	11	15.2452	0.0696558
pr_nibble	europe_osm	12	20.067	0.0529186
pr_nibble	europe_osm	13	15.8651	0.0669342
pr_nibble	europe_osm	14	16.1729	0.0656603
pr_nibble	europe_osm	15	15.8811	0.0668668
pr_nibble	europe_osm	16	17.041	0.0623155
pr_nibble	hollywood-2009	1	1.60599	1
pr_nibble	hollywood-2009	2	2.70987	0.592645
pr_nibble	hollywood-2009	3	12.5859	0.127602
pr_nibble	hollywood-2009	4	12.955	0.123967
pr_nibble	hollywood-2009	5	13.1569	0.122064
pr_nibble	hollywood-2009	6	13.103	0.122566
pr_nibble	hollywood-2009	7	12.677	0.126686
pr_nibble	hollywood-2009	8	5.22709	0.307243
pr_nibble	hollywood-2009	9	13.6061	0.118035
pr_nibble	hollywood-2009	10	5.13005	0.313055
pr_nibble	hollywood-2009	11	14.6151	0.109886
pr_nibble	hollywood-2009	12	16.036	0.100149
pr_nibble	hollywood-2009	13	15.0352	0.106816
pr_nibble	hollywood-2009	14	17.1752	0.0935062
pr_nibble	hollywood-2009	15	15.101	0.10635
pr_nibble	hollywood-2009	16	14.662	0.109534

Table 10. Tabular Data for Project

primitive	dataset	num-gpus	avg-process-time	speedup
proj	ml_1000000	1	118.109	1
proj	ml_1000000	2	109.162	1.08196
proj	ml_1000000	3	100.278	1.17781
proj	ml_1000000	4	93.5086	1.26308
proj	ml_1000000	5	98.6481	1.19728
proj	ml_1000000	6	168.59	0.700569
proj	ml_1000000	7	158.302	0.746099
proj	ml_1000000	8	182.951	0.645579
proj	ml_1000000	9	176.637	0.668655
proj	ml_1000000	10	141.592	0.834154
proj	ml_1000000	11	128.888	0.916372
proj	ml_1000000	12	188.994	0.624938
proj	ml_1000000	13	200.228	0.589874
proj	ml_1000000	14	179.094	0.659483
proj	ml_1000000	15	334.797	0.352779
proj	ml_1000000	16	244.981	0.482116
proj	ml_5000000	1	679.568	1
proj	ml_5000000	2	541.636	1.25466
proj	ml_5000000	3	507.644	1.33867
proj	ml_5000000	4	481.724	1.4107
proj	ml_5000000	5	450.971	1.5069
proj	ml_5000000	6	470.028	1.4458
proj	ml_5000000	7	519.952	1.30698
proj	ml_5000000	8	454.093	1.49654
proj	ml_5000000	9	532.999	1.27499
proj	ml_5000000	10	477.406	1.42346
proj	ml_5000000	11	450.121	1.50975
proj	ml_5000000	12	472.946	1.43688
proj	ml_5000000	13	478.281	1.42086
proj	ml_5000000	14	451.592	1.50483
proj	ml_5000000	15	489.375	1.38865
proj	ml_5000000	16	509.011	1.33508
proj	ml_full	1	2318.65	1
proj	ml_full	2	2965.21	0.78195
proj	ml_full	3	2193.6	1.05701

proj	ml_full	4	2095.3	1.1066
proj	ml_full	5	2059.51	1.12582
proj	ml_full	6	2018.83	1.14851
proj	ml_full	7	1944.38	1.19248
proj	ml_full	8	1913.56	1.21169
proj	ml_full	9	1859.59	1.24686
proj	ml_full	10	1800.46	1.28781
proj	ml_full	11	1807.37	1.28288
proj	ml_full	12	1801.11	1.28734
proj	ml_full	13	1790.01	1.29532
proj	ml_full	14	1780.89	1.30196
proj	ml_full	15	1829.09	1.26765
proj	ml_full	16	1794.11	1.29237

Table 11. Tabular Data for rw_directed-greedy

primitive	dataset	variant	num-gpus	avg-process-time	speedup
rw	dir_gs_twitter	directed-greedy	1	39.4513	1
rw	dir_gs_twitter	directed-greedy	2	41.3393	0.954329
rw	dir_gs_twitter	directed-greedy	3	33.4494	1.17943
rw	dir_gs_twitter	directed-greedy	4	37.0721	1.06418
rw	dir_gs_twitter	directed-greedy	5	29.9544	1.31704
rw	dir_gs_twitter	directed-greedy	6	33.4877	1.17808
rw	dir_gs_twitter	directed-greedy	7	31.2809	1.26119
rw	dir_gs_twitter	directed-greedy	8	33.5908	1.17447
rw	dir_gs_twitter	directed-greedy	9	32.8	1.20278
rw	dir_gs_twitter	directed-greedy	10	31.3871	1.25693
rw	dir_gs_twitter	directed-greedy	11	30.2808	1.30285
rw	dir_gs_twitter	directed-greedy	12	39.1222	1.00841
rw	dir_gs_twitter	directed-greedy	13	38.2514	1.03137
rw	dir_gs_twitter	directed-greedy	14	42.3774	0.93095
rw	dir_gs_twitter	directed-greedy	15	41.7984	0.943847
rw	dir_gs_twitter	directed-greedy	16	37.2768	1.05833

Table 12. Tabular Data for rw_directed-uniform

primitive	dataset	variant	num-gpus	avg-process-time	speedup
rw	dir_gs_twitter	directed-uniform	1	18.5425	1
rw	dir_gs_twitter	directed-uniform	2	68.5254	0.270593
rw	dir_gs_twitter	directed-uniform	3	78.0548	0.237558
rw	dir_gs_twitter	directed-uniform	4	87.2627	0.212491
rw	dir_gs_twitter	directed-uniform	5	86.3617	0.214708
rw	dir_gs_twitter	directed-uniform	6	105.554	0.175669
rw	dir_gs_twitter	directed-uniform	7	106.176	0.174639
rw	dir_gs_twitter	directed-uniform	8	115.794	0.160134
rw	dir_gs_twitter	directed-uniform	9	129.298	0.143409
rw	dir_gs_twitter	directed-uniform	10	122.168	0.151778
rw	dir_gs_twitter	directed-uniform	11	124.95	0.148399
rw	dir_gs_twitter	directed-uniform	12	154.088	0.120337
rw	dir_gs_twitter	directed-uniform	13	153.653	0.120678
rw	dir_gs_twitter	directed-uniform	14	149.857	0.123734
rw	dir_gs_twitter	directed-uniform	15	163.669	0.113293
rw	dir_gs_twitter	directed-uniform	16	167.475	0.110718

Table 13. Tabular Data for rw_undirected-uniform

primitive	dataset	variant	num-gpus	avg-process-time	speedup
rw	dir_gs_twitter	undirected-uniform	1	484.335	1
rw	dir_gs_twitter	undirected-uniform	2	784.393	0.617464
rw	dir_gs_twitter	undirected-uniform	3	731.489	0.662121
rw	dir_gs_twitter	undirected-uniform	4	678.455	0.713878
rw	dir_gs_twitter	undirected-uniform	5	653.389	0.741265
rw	dir_gs_twitter	undirected-uniform	6	630.627	0.768021
rw	dir_gs_twitter	undirected-uniform	7	606.658	0.798365
rw	dir_gs_twitter	undirected-uniform	8	702.976	0.688977
rw	dir_gs_twitter	undirected-uniform	9	637.977	0.759172
rw	dir_gs_twitter	undirected-uniform	10	749.448	0.646255
rw	dir_gs_twitter	undirected-uniform	11	639.487	0.75738
rw	dir_gs_twitter	undirected-uniform	12	698.939	0.692957
rw	dir_gs_twitter	undirected-uniform	13	750.53	0.645323
rw	dir_gs_twitter	undirected-uniform	14	636.293	0.761181
rw	dir_gs_twitter	undirected-uniform	15	761.795	0.635781
rw	dir_gs_twitter	undirected-uniform	16	831.94	0.582175

Table 14. Tabular Data for sgm

primitive	dataset	num-gpus	avg-process-time	speedup
sgm	DS00833	1	17.8509	1
sgm	DS00833	4	7.08077	2.52104
sgm	DS00833	5	9.91024	1.80126
sgm	DS00833	6	9.10208	1.96119
sgm	DS00833	7	5.92381	3.01341
sgm	DS00833	8	5.13466	3.47655
sgm	DS00833	9	8.7584	2.03814
sgm	DS00833	10	8.63677	2.06685
sgm	DS00833	11	281.119	0.0634994
sgm	DS00833	12	6.04387	2.95355
sgm	DS00833	13	6.41021	2.78476
sgm	DS00833	14	7.39926	2.41252
sgm	DS00833	15	7.05302	2.53095
sgm	DS00833	16	6.30525	2.83111
sgm	DS01216	1	30.6063	1
sgm	DS01216	4	11.1936	2.73426
sgm	DS01216	5	10.4081	2.94064
sgm	DS01216	6	8.55526	3.57749
sgm	DS01216	7	12.2431	2.49988
sgm	DS01216	8	9.62896	3.17857
sgm	DS01216	9	9.91882	3.08568
sgm	DS01216	10	7.86448	3.89172
sgm	DS01216	11	5.11059	5.9888
sgm	DS01216	12	6.26483	4.88542
sgm	DS01216	13	10.438	2.93222
sgm	DS01216	14	10.2681	2.98072
sgm	DS01216	15	17.1682	1.78273
sgm	DS01216	16	7.00909	4.36666
sgm	DS01876	1	63.8648	1
sgm	DS01876	4	19.3141	3.30664
sgm	DS01876	5	16.0728	3.97347
sgm	DS01876	6	15.455	4.13232
sgm	DS01876	7	17.3419	3.68269
sgm	DS01876	8	13.0148	4.90707
sgm	DS01876	9	11.4081	5.59819

sgm	DS01876	10	16.1354	3.95806
sgm	DS01876	11	15.2498	4.18792
sgm	DS01876	12	13.1866	4.84318
sgm	DS01876	13	13.192	4.84118
sgm	DS01876	14	12.5347	5.09505
sgm	DS01876	15	10.8288	5.89767
sgm	DS01876	16	10.2508	6.23024
sgm	DS03231	1	188.603	1
sgm	DS03231	5	37.6289	5.01217
sgm	DS03231	6	33.7132	5.59434
sgm	DS03231	9	21.5498	8.75194
sgm	DS03231	10	21.8756	8.62161
sgm	DS03231	11	21.4444	8.79498
sgm	DS03231	12	25.7042	7.33743
sgm	DS03231	13	19.7032	9.5722
sgm	DS03231	14	18.6575	10.1087
sgm	DS03231	15	16.3844	11.5111
sgm	DS03231	16	17.3342	10.8804
sgm	DS06481	1	1078.58	1
sgm	DS06481	5	134.765	8.00344
sgm	DS06481	6	107.924	9.99389
sgm	DS06481	9	71.1115	15.1675
sgm	DS06481	10	71.899	15.0014
sgm	DS06481	11	56.5601	19.0697
sgm	DS06481	12	61.5752	17.5165
sgm	DS06481	16	50.6436	21.2975
sgm	DS16784	1	4889.5	1
sgm	DS16784	2	12452.4	0.392657
sgm	DS16784	4	2901.11	1.68539
sgm	DS16784	8	855.194	5.71742
sgm	DS16784	10	595.373	8.2125
sgm	DS16784	11	486.601	10.0483
sgm	DS16784	12	441.323	11.0792
sgm	DS16784	13	414.08	11.8081
sgm	DS16784	14	358.727	13.6302
sgm	DS16784	15	324.382	15.0733
sgm	DS16784	16	326.032	14.997

Table 15. Tabular Data for vn_num_seeds-10

primitive	dataset	variant	num-gpus	avg-process-time	speedup
vn	enron	num_seeds-10	1	0.182	1
vn	enron	num_seeds-10	2	0.523	0.347992
vn	enron	num_seeds-10	3	0.464	0.392241
vn	enron	num_seeds-10	4	1.112	0.163669
vn	enron	num_seeds-10	5	0.761	0.239159
vn	enron	num_seeds-10	6	1.039	0.175168
vn	enron	num_seeds-10	7	1.39	0.130935
vn	enron	num_seeds-10	8	1.689	0.107756
vn	enron	num_seeds-10	9	0.837	0.217443
vn	enron	num_seeds-10	10	0.986	0.184584
vn	enron	num_seeds-10	11	1.151	0.158123
vn	enron	num_seeds-10	12	1.876	0.0970149
vn	enron	num_seeds-10	13	2.109	0.0862968
vn	enron	num_seeds-10	14	2.305	0.0789588
vn	enron	num_seeds-10	15	2.509	0.0725389
vn	enron	num_seeds-10	16	2.938	0.0619469
vn	hollywood-2009	num_seeds-10	1	12.241	1
vn	hollywood-2009	num_seeds-10	2	8.974	1.36405
vn	hollywood-2009	num_seeds-10	3	7.629	1.60454
vn	hollywood-2009	num_seeds-10	4	7.48	1.6365
vn	hollywood-2009	num_seeds-10	5	7.531	1.62541
vn	hollywood-2009	num_seeds-10	6	7.2	1.70014
vn	hollywood-2009	num_seeds-10	7	9.245	1.32407
vn	hollywood-2009	num_seeds-10	8	7.757	1.57806
vn	hollywood-2009	num_seeds-10	9	8.141	1.50362
vn	hollywood-2009	num_seeds-10	10	9.407	1.30127
vn	hollywood-2009	num_seeds-10	11	8.991	1.36147
vn	hollywood-2009	num_seeds-10	12	21.411	0.571715
vn	hollywood-2009	num_seeds-10	13	10.928	1.12015
vn	hollywood-2009	num_seeds-10	14	11.387	1.075
vn	hollywood-2009	num_seeds-10	15	11.042	1.10859
vn	hollywood-2009	num_seeds-10	16	11.441	1.06992
vn	indochina-2004	num_seeds-10	1	59.344	1
vn	indochina-2004	num_seeds-10	2	50.935	1.16509
vn	indochina-2004	num_seeds-10	3	45.277	1.31069

vn	indochina-2004	num_seeds-10	4	44.234	1.34159
vn	indochina-2004	num_seeds-10	5	50.41	1.17723
vn	indochina-2004	num_seeds-10	6	45.299	1.31005
vn	indochina-2004	num_seeds-10	7	51.977	1.14174
vn	indochina-2004	num_seeds-10	8	66.553	0.89168
vn	indochina-2004	num_seeds-10	9	47.16	1.25835
vn	indochina-2004	num_seeds-10	10	50.722	1.16999
vn	indochina-2004	num_seeds-10	11	52.984	1.12004
vn	indochina-2004	num_seeds-10	12	49.287	1.20405
vn	indochina-2004	num_seeds-10	13	54.144	1.09604
vn	indochina-2004	num_seeds-10	14	52.486	1.13066
vn	indochina-2004	num_seeds-10	15	53.795	1.10315
vn	indochina-2004	num_seeds-10	16	63.273	0.937904
vn	rmat18	num_seeds-10	1	1.289	1
vn	rmat18	num_seeds-10	2	1.548	0.832687
vn	rmat18	num_seeds-10	3	1.668	0.772782
vn	rmat18	num_seeds-10	4	1.797	0.717307
vn	rmat18	num_seeds-10	5	1.862	0.692266
vn	rmat18	num_seeds-10	6	1.819	0.708631
vn	rmat18	num_seeds-10	7	2.051	0.628474
vn	rmat18	num_seeds-10	8	2.17	0.594009
vn	rmat18	num_seeds-10	9	2.018	0.638751
vn	rmat18	num_seeds-10	10	2.41	0.534855
vn	rmat18	num_seeds-10	11	2.534	0.508682
vn	rmat18	num_seeds-10	12	2.739	0.47061
vn	rmat18	num_seeds-10	13	2.963	0.435032
vn	rmat18	num_seeds-10	14	3.08	0.418506
vn	rmat18	num_seeds-10	15	3.475	0.370935
vn	rmat18	num_seeds-10	16	3.13	0.411821
vn	rmat20	num_seeds-10	1	4.047	1
vn	rmat20	num_seeds-10	2	3.572	1.13298
vn	rmat20	num_seeds-10	3	3.191	1.26825
vn	rmat20	num_seeds-10	4	3.358	1.20518
vn	rmat20	num_seeds-10	5	3.098	1.30633
vn	rmat20	num_seeds-10	6	3.152	1.28395
vn	rmat20	num_seeds-10	7	3.273	1.23648
vn	rmat20	num_seeds-10	8	3.213	1.25957

vn	rmat20	num_seeds-10	9	3.29	1.23009
vn	rmat20	num_seeds-10	10	3.578	1.13108
vn	rmat20	num_seeds-10	11	3.463	1.16864
vn	rmat20	num_seeds-10	12	3.992	1.01378
vn	rmat20	num_seeds-10	13	4.122	0.981805
vn	rmat20	num_seeds-10	14	4.416	0.91644
vn	rmat20	num_seeds-10	15	4.175	0.969341
vn	rmat20	num_seeds-10	16	4.684	0.864005
vn	rmat22	num_seeds-10	1	17.277	1
vn	rmat22	num_seeds-10	2	12.951	1.33403
vn	rmat22	num_seeds-10	3	10.74	1.60866
vn	rmat22	num_seeds-10	4	9.054	1.90822
vn	rmat22	num_seeds-10	5	8.464	2.04123
vn	rmat22	num_seeds-10	6	7.718	2.23853
vn	rmat22	num_seeds-10	7	7.7	2.24377
vn	rmat22	num_seeds-10	8	8.281	2.08634
vn	rmat22	num_seeds-10	9	7.05	2.45064
vn	rmat22	num_seeds-10	10	7.083	2.43922
vn	rmat22	num_seeds-10	11	7.394	2.33662
vn	rmat22	num_seeds-10	12	7.576	2.28049
vn	rmat22	num_seeds-10	13	8.166	2.11572
vn	rmat22	num_seeds-10	14	8.919	1.9371
vn	rmat22	num_seeds-10	15	9.576	1.8042
vn	rmat22	num_seeds-10	16	9.432	1.83174
vn	rmat24	num_seeds-10	1	136.888	1
vn	rmat24	num_seeds-10	2	76.379	1.79222
vn	rmat24	num_seeds-10	3	59.317	2.30774
vn	rmat24	num_seeds-10	4	48.33	2.83236
vn	rmat24	num_seeds-10	5	42.359	3.23162
vn	rmat24	num_seeds-10	6	37.052	3.69448
vn	rmat24	num_seeds-10	7	37.085	3.6912
vn	rmat24	num_seeds-10	8	32.071	4.26828
vn	rmat24	num_seeds-10	9	29.714	4.60685
vn	rmat24	num_seeds-10	10	29.338	4.66589
vn	rmat24	num_seeds-10	11	37.358	3.66422
vn	rmat24	num_seeds-10	12	29.001	4.72011
vn	rmat24	num_seeds-10	13	28.306	4.83601

vn	rmat24	num_seeds-10	14	28.242	4.84697
vn	rmat24	num_seeds-10	15	26.05	5.25482
vn	rmat24	num_seeds-10	16	32.828	4.16986

Table 16 Tabular Data for vn_num_seeds-100

primitive	dataset	variant	num-gpus	avg-process-time	speedup
vn	enron	num_seeds-100	1	0.378	1
vn	enron	num_seeds-100	2	1.006	0.375746
vn	enron	num_seeds-100	3	1.008	0.375
vn	enron	num_seeds-100	4	1.561	0.242152
vn	enron	num_seeds-100	5	1.894	0.199578
vn	enron	num_seeds-100	6	1.982	0.190716
vn	enron	num_seeds-100	7	2.119	0.178386
vn	enron	num_seeds-100	8	3.154	0.119848
vn	enron	num_seeds-100	9	2.028	0.186391
vn	enron	num_seeds-100	10	2.152	0.175651
vn	enron	num_seeds-100	11	2.457	0.153846
vn	enron	num_seeds-100	12	3.271	0.115561
vn	enron	num_seeds-100	13	3.722	0.101558
vn	enron	num_seeds-100	14	3.916	0.0965271
vn	enron	num_seeds-100	15	3.868	0.0977249
vn	enron	num_seeds-100	16	4.686	0.0806658
vn	hollywood-2009	num_seeds-100	1	11.579	1
vn	hollywood-2009	num_seeds-100	2	9.049	1.27959
vn	hollywood-2009	num_seeds-100	3	7.02	1.64943
vn	hollywood-2009	num_seeds-100	4	7.839	1.4771
vn	hollywood-2009	num_seeds-100	5	6.879	1.68324
vn	hollywood-2009	num_seeds-100	6	7.229	1.60174
vn	hollywood-2009	num_seeds-100	7	7.229	1.60174
vn	hollywood-2009	num_seeds-100	8	8.416	1.37583
vn	hollywood-2009	num_seeds-100	9	7.631	1.51736
vn	hollywood-2009	num_seeds-100	10	8.813	1.31385
vn	hollywood-2009	num_seeds-100	11	9.23	1.2545
vn	hollywood-2009	num_seeds-100	12	9.78	1.18395
vn	hollywood-2009	num_seeds-100	13	10.073	1.14951
vn	hollywood-2009	num_seeds-100	14	10.115	1.14474
vn	hollywood-2009	num_seeds-100	15	9.625	1.20301
vn	hollywood-2009	num_seeds-100	16	10.815	1.07064
vn	indochina-2004	num_seeds-100	1	56.788	1
vn	indochina-2004	num_seeds-100	2	54.691	1.03834
vn	indochina-2004	num_seeds-100	3	44.627	1.2725

vn	indochina-2004	num_seeds-100	4	44.226	1.28404
vn	indochina-2004	num_seeds-100	5	52.462	1.08246
vn	indochina-2004	num_seeds-100	6	46.112	1.23152
vn	indochina-2004	num_seeds-100	7	47.633	1.1922
vn	indochina-2004	num_seeds-100	8	46.315	1.22613
vn	indochina-2004	num_seeds-100	9	48.347	1.17459
vn	indochina-2004	num_seeds-100	10	51.368	1.10551
vn	indochina-2004	num_seeds-100	11	52.947	1.07254
vn	indochina-2004	num_seeds-100	12	49.687	1.14291
vn	indochina-2004	num_seeds-100	13	50.333	1.12825
vn	indochina-2004	num_seeds-100	14	52.319	1.08542
vn	indochina-2004	num_seeds-100	15	54.403	1.04384
vn	indochina-2004	num_seeds-100	16	56.253	1.00951
vn	rmat18	num_seeds-100	1	1.281	1
vn	rmat18	num_seeds-100	2	1.591	0.805154
vn	rmat18	num_seeds-100	3	1.806	0.709302
vn	rmat18	num_seeds-100	4	1.75	0.732
vn	rmat18	num_seeds-100	5	1.751	0.731582
vn	rmat18	num_seeds-100	6	1.816	0.705396
vn	rmat18	num_seeds-100	7	2.692	0.475854
vn	rmat18	num_seeds-100	8	2.388	0.536432
vn	rmat18	num_seeds-100	9	2.05	0.624878
vn	rmat18	num_seeds-100	10	2.487	0.515078
vn	rmat18	num_seeds-100	11	2.305	0.555748
vn	rmat18	num_seeds-100	12	2.824	0.453612
vn	rmat18	num_seeds-100	13	2.6	0.492692
vn	rmat18	num_seeds-100	14	3.217	0.398197
vn	rmat18	num_seeds-100	15	2.932	0.436903
vn	rmat18	num_seeds-100	16	3.212	0.398817
vn	rmat20	num_seeds-100	1	4.138	1
vn	rmat20	num_seeds-100	2	3.373	1.2268
vn	rmat20	num_seeds-100	3	3.158	1.31032
vn	rmat20	num_seeds-100	4	3.322	1.24564
vn	rmat20	num_seeds-100	5	2.815	1.46998
vn	rmat20	num_seeds-100	6	3.065	1.35008
vn	rmat20	num_seeds-100	7	4.402	0.940027
vn	rmat20	num_seeds-100	8	3.348	1.23596

vn	rmat20	num_seeds-100	9	3.339	1.23929
vn	rmat20	num_seeds-100	10	3.585	1.15425
vn	rmat20	num_seeds-100	11	3.749	1.10376
vn	rmat20	num_seeds-100	12	3.941	1.04999
vn	rmat20	num_seeds-100	13	4.18	0.989952
vn	rmat20	num_seeds-100	14	4.454	0.929053
vn	rmat20	num_seeds-100	15	4.445	0.930934
vn	rmat20	num_seeds-100	16	4.783	0.865147
vn	rmat22	num_seeds-100	1	16.675	1
vn	rmat22	num_seeds-100	2	11.839	1.40848
vn	rmat22	num_seeds-100	3	9.592	1.73843
vn	rmat22	num_seeds-100	4	8.706	1.91535
vn	rmat22	num_seeds-100	5	8.646	1.92864
vn	rmat22	num_seeds-100	6	7.992	2.08646
vn	rmat22	num_seeds-100	7	7.661	2.17661
vn	rmat22	num_seeds-100	8	7.043	2.3676
vn	rmat22	num_seeds-100	9	7.42	2.2473
vn	rmat22	num_seeds-100	10	7.129	2.33904
vn	rmat22	num_seeds-100	11	7.478	2.22987
vn	rmat22	num_seeds-100	12	7.454	2.23705
vn	rmat22	num_seeds-100	13	8.627	1.93289
vn	rmat22	num_seeds-100	14	9.432	1.76792
vn	rmat22	num_seeds-100	15	9.254	1.80192
vn	rmat22	num_seeds-100	16	11.77	1.41674
vn	rmat24	num_seeds-100	1	137.55	1
vn	rmat24	num_seeds-100	2	79.82	1.72325
vn	rmat24	num_seeds-100	3	64.532	2.1315
vn	rmat24	num_seeds-100	4	47.899	2.87167
vn	rmat24	num_seeds-100	5	45.012	3.05585
vn	rmat24	num_seeds-100	6	37.821	3.63687
vn	rmat24	num_seeds-100	7	35.677	3.85543
vn	rmat24	num_seeds-100	8	31.968	4.30274
vn	rmat24	num_seeds-100	9	29.807	4.61469
vn	rmat24	num_seeds-100	10	29.747	4.624
vn	rmat24	num_seeds-100	11	28.378	4.84706
vn	rmat24	num_seeds-100	12	28.638	4.80306
vn	rmat24	num_seeds-100	13	28.018	4.90934

vn	rmat24	num_seeds-100	14	28.053	4.90322
vn	rmat24	num_seeds-100	15	26.453	5.19979
vn	rmat24	num_seeds-100	16	25.779	5.33574

Table 17. Tabular Data for vn_num_seeds-1000

primitive	dataset	variant	num-gpus	avg-process-time	speedup
vn	enron	num_seeds-1000	1	0.364	1
vn	enron	num_seeds-1000	2	1.028	0.354086
vn	enron	num_seeds-1000	3	1.09	0.333945
vn	enron	num_seeds-1000	4	1.435	0.253659
vn	enron	num_seeds-1000	5	2.13	0.170892
vn	enron	num_seeds-1000	6	2.235	0.162864
vn	enron	num_seeds-1000	7	2.551	0.142689
vn	enron	num_seeds-1000	8	2.363	0.154041
vn	enron	num_seeds-1000	9	2.035	0.17887
vn	enron	num_seeds-1000	10	2.173	0.16751
vn	enron	num_seeds-1000	11	2.218	0.164112
vn	enron	num_seeds-1000	12	3.182	0.114393
vn	enron	num_seeds-1000	13	3.66	0.0994536
vn	enron	num_seeds-1000	14	4.009	0.0907957
vn	enron	num_seeds-1000	15	4.197	0.0867286
vn	enron	num_seeds-1000	16	4.596	0.0791993
vn	hollywood-2009	num_seeds-1000	1	11.679	1
vn	hollywood-2009	num_seeds-1000	2	8.915	1.31004
vn	hollywood-2009	num_seeds-1000	3	7.028	1.66178
vn	hollywood-2009	num_seeds-1000	4	7.084	1.64864
vn	hollywood-2009	num_seeds-1000	5	6.548	1.7836
vn	hollywood-2009	num_seeds-1000	6	6.562	1.77979
vn	hollywood-2009	num_seeds-1000	7	6.579	1.77519
vn	hollywood-2009	num_seeds-1000	8	6.147	1.89995
vn	hollywood-2009	num_seeds-1000	9	7.42	1.57399
vn	hollywood-2009	num_seeds-1000	10	8.785	1.32943
vn	hollywood-2009	num_seeds-1000	11	7.808	1.49577
vn	hollywood-2009	num_seeds-1000	12	9.638	1.21177
vn	hollywood-2009	num_seeds-1000	13	9.873	1.18292
vn	hollywood-2009	num_seeds-1000	14	10.109	1.15531
vn	hollywood-2009	num_seeds-1000	15	10.387	1.12439
vn	hollywood-2009	num_seeds-1000	16	11.295	1.034
vn	indochina-2004	num_seeds-1000	1	55.18	1
vn	indochina-2004	num_seeds-1000	2	51.016	1.08162
vn	indochina-2004	num_seeds-1000	3	45.466	1.21365

vn	indochina-2004	num_seeds-1000	4	47.048	1.17284
vn	indochina-2004	num_seeds-1000	5	48.808	1.13055
vn	indochina-2004	num_seeds-1000	6	55.702	0.990629
vn	indochina-2004	num_seeds-1000	7	48.728	1.13241
vn	indochina-2004	num_seeds-1000	8	83.595	0.660087
vn	indochina-2004	num_seeds-1000	9	47.462	1.16261
vn	indochina-2004	num_seeds-1000	10	49.971	1.10424
vn	indochina-2004	num_seeds-1000	11	51.083	1.0802
vn	indochina-2004	num_seeds-1000	12	48.547	1.13663
vn	indochina-2004	num_seeds-1000	13	49.751	1.10912
vn	indochina-2004	num_seeds-1000	14	50.488	1.09293
vn	indochina-2004	num_seeds-1000	15	53.265	1.03595
vn	indochina-2004	num_seeds-1000	16	54.336	1.01553
vn	rmat18	num_seeds-1000	1	1.279	1
vn	rmat18	num_seeds-1000	2	1.504	0.850399
vn	rmat18	num_seeds-1000	3	1.958	0.653218
vn	rmat18	num_seeds-1000	4	1.779	0.718943
vn	rmat18	num_seeds-1000	5	1.781	0.718136
vn	rmat18	num_seeds-1000	6	1.843	0.693977
vn	rmat18	num_seeds-1000	7	1.972	0.64858
vn	rmat18	num_seeds-1000	8	2.842	0.450035
vn	rmat18	num_seeds-1000	9	2.292	0.558028
vn	rmat18	num_seeds-1000	10	2.402	0.532473
vn	rmat18	num_seeds-1000	11	2.33	0.548927
vn	rmat18	num_seeds-1000	12	2.858	0.447516
vn	rmat18	num_seeds-1000	13	2.666	0.479745
vn	rmat18	num_seeds-1000	14	3.079	0.415395
vn	rmat18	num_seeds-1000	15	3.248	0.393781
vn	rmat18	num_seeds-1000	16	3.154	0.405517
vn	rmat20	num_seeds-1000	1	4.069	1
vn	rmat20	num_seeds-1000	2	3.277	1.24168
vn	rmat20	num_seeds-1000	3	2.826	1.43984
vn	rmat20	num_seeds-1000	4	3.183	1.27835
vn	rmat20	num_seeds-1000	5	3.06	1.32974
vn	rmat20	num_seeds-1000	6	3.083	1.31982
vn	rmat20	num_seeds-1000	7	3.087	1.31811
vn	rmat20	num_seeds-1000	8	4.985	0.816249

vn	rmat20	num_seeds-1000	9	3.303	1.23191
vn	rmat20	num_seeds-1000	10	3.34	1.21826
vn	rmat20	num_seeds-1000	11	3.673	1.10781
vn	rmat20	num_seeds-1000	12	3.961	1.02727
vn	rmat20	num_seeds-1000	13	4.171	0.975545
vn	rmat20	num_seeds-1000	14	4.364	0.932401
vn	rmat20	num_seeds-1000	15	4.646	0.875807
vn	rmat20	num_seeds-1000	16	4.623	0.880164
vn	rmat22	num_seeds-1000	1	17.182	1
vn	rmat22	num_seeds-1000	2	12.098	1.42023
vn	rmat22	num_seeds-1000	3	9.511	1.80654
vn	rmat22	num_seeds-1000	4	8.691	1.97699
vn	rmat22	num_seeds-1000	5	8.311	2.06738
vn	rmat22	num_seeds-1000	6	7.64	2.24895
vn	rmat22	num_seeds-1000	7	7.552	2.27516
vn	rmat22	num_seeds-1000	8	7.586	2.26496
vn	rmat22	num_seeds-1000	9	7.082	2.42615
vn	rmat22	num_seeds-1000	10	7.289	2.35725
vn	rmat22	num_seeds-1000	11	7.33	2.34407
vn	rmat22	num_seeds-1000	12	7.466	2.30137
vn	rmat22	num_seeds-1000	13	7.959	2.15881
vn	rmat22	num_seeds-1000	14	8.088	2.12438
vn	rmat22	num_seeds-1000	15	9.495	1.80958
vn	rmat22	num_seeds-1000	16	9.434	1.82128
vn	rmat24	num_seeds-1000	1	138.951	1
vn	rmat24	num_seeds-1000	2	79.828	1.74063
vn	rmat24	num_seeds-1000	3	60.35	2.30242
vn	rmat24	num_seeds-1000	4	48.995	2.83602
vn	rmat24	num_seeds-1000	5	44.683	3.10971
vn	rmat24	num_seeds-1000	6	37.56	3.69944
vn	rmat24	num_seeds-1000	7	40.524	3.42886
vn	rmat24	num_seeds-1000	8	32.029	4.33829
vn	rmat24	num_seeds-1000	9	30.584	4.54326
vn	rmat24	num_seeds-1000	10	29.655	4.68558
vn	rmat24	num_seeds-1000	11	29.224	4.75469
vn	rmat24	num_seeds-1000	12	28.501	4.8753
vn	rmat24	num_seeds-1000	13	39.015	3.56148

vn	rmat24	num_seeds-1000	14	27.993	4.96378
vn	rmat24	num_seeds-1000	15	26.284	5.28652
vn	rmat24	num_seeds-1000	16	25.696	5.4075

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	**DESCRIPTION**
AC	Application Classification, a HIVE app
CUDA	Compute Unified Device Architecture, NVIDIA’s GPU Programming Environment
DARPA	Defense Advanced Research Projects Agency
DIR	Directory
GPU	Graphics Processing Unit
GS	GraphSAGE, a HIVE App
GTF	Graph Trend Filtering, a HIVE App
HIVE	Hierarchical Identify Verify Exploit, a DARPA Program
HTML	HyperText Markup Language
LGC	Local Graph Clustering, a HIVE App
OUTPUT	The Output Directory, a Destination for Writing Output
SFL	Sparse Fused Lasso, a HIVE App
SGM	Seeded Graph Matching, a HIVE App
SLURM	Simple Linux Utility for Resource Management, a Job Scheduler for Running Work
SpGEMM	Sparse General Matrix Multiply (multiplying two sparse matrices together)
SSSP	Single Source Shortest Path, a Graph Computation
TC	Triangle Counting, a Graph Computation
UC	University of California