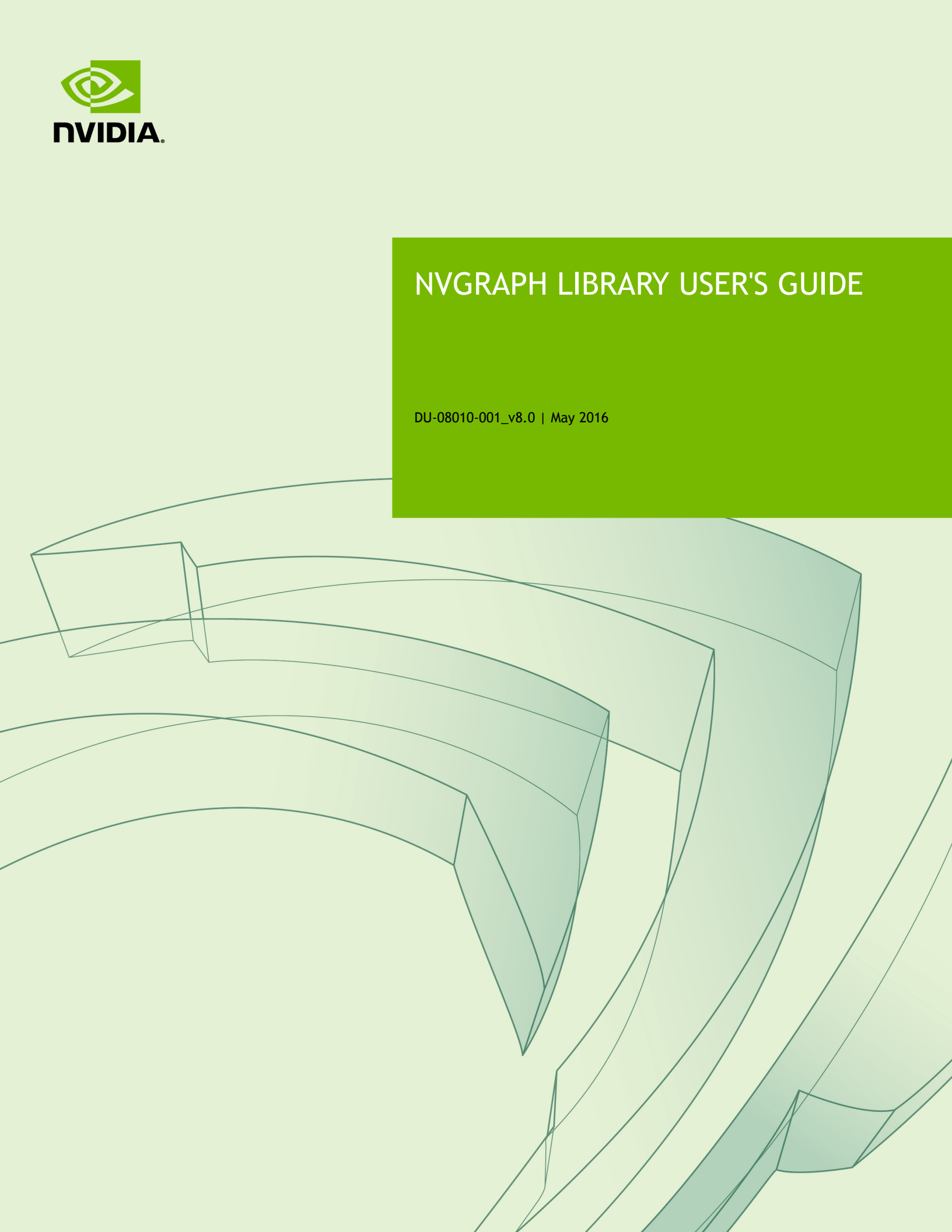

NVGRAPH LIBRARY USER'S GUIDE

DU-08010-001_v8.0 | May 2016

TABLE OF CONTENTS

Chapter 1. Introduction..... 1
Chapter 2. nvGRAPH API Reference..... 2
2.1. Return value nvgraphStatus_t..... 2
2.2. nvGRAPH graph topology types..... 3
2.3. nvGRAPH topology structure types..... 3
nvgraphCSRTopology32I_t..... 4
nvgraphCSCTopology32I_t..... 4
2.4. Function nvgraphCreate()..... 5
2.5. Function nvgraphDestroy()..... 5
2.6. Function nvgraphCreateGraphDescr()..... 5
2.7. Function nvgraphDestroyGraphDescr()..... 6
2.8. Function nvgraphSetGraphStructure()..... 6
2.9. Function nvgraphGetGraphStructure()..... 7
2.10. Function nvgraphAllocateEdgeData()..... 8
2.11. Function nvgraphSetEdgeData()..... 9
2.12. Function nvgraphGetEdgeData()..... 9
2.13. Function nvgraphAllocateVertexData()..... 10
2.14. Function nvgraphSetVertexData()..... 11
2.15. Function nvgraphGetVertexData()..... 11
2.16. Function nvgraphExtractSubgraphByVertex()..... 12
2.17. Function nvgraphExtractSubgraphByEdge()..... 13
2.18. Function nvgraphWidestPath()..... 13
2.19. Function nvgraphSssp()..... 14
2.20. Function nvgraphSrSpmv()..... 15
2.21. Function nvgraphPagerank()..... 17
2.22. Function nvgraphStatusGetString()..... 18
Chapter 3. nvGRAPH Code Examples..... 20
3.1. nvGRAPH pagerank example..... 21
3.2. nvGRAPH SSSP example..... 22
3.3. nvGRAPH Semi-Ring SPMV example..... 23

Chapter 1. INTRODUCTION

Many advanced data analytics problems can be couched as graph problems. In turn, many of the common graph analytics used today can be couched as sparse linear algebra. This is the motivation for nvGRAPH, new in NVIDIA® CUDA™ 8.0, which harnesses the power of GPUs for linear algebra to handle the largest graph analytics and big data analytics problems.

This first release provides basic graph construction and manipulation primitives, and a set of useful graph algorithms optimized for the GPU. The core functionality is a SPMV (sparse matrix vector product) using a semi-ring model with automatic load balancing for any sparsity pattern. For more information on semi-rings and their uses, we recommend the book "Graph Algorithms in the Language of linear algebra", by Jeremy Kepner and John Gilbert.

A typical workflow for using nvGRAPH is begin by calling nvgraphCreate() to initialize the library. Next the user can proceed to upload graph data to the library through nvGRAPH's API, or if there is already a graph loaded in device memory then provide a device pointer to this data. Then the user can execute graph algorithms on the data, extract a subgraph from the data, or reformat the data using the nvGRAPH API. The user can download the results back to host, or copy them to another location on the device, and once all calculations are done the user should call nvgraphDestroy() to free resources used by nvGRAPH.

Chapter 2. NVGRAPH API REFERENCE

This chapter specifies the behavior of the nvGRAPH library functions by describing their input/output parameters, data types, and error codes.

2.1. Return value nvgraphStatus_t

All nvGRAPH Library return values except for **NVGRAPH_STATUS_SUCCESS** indicate that the current API call failed and the user should reconfigure to correct the problem. The possible return values are defined as follows:

Return Values

NVGRAPH_STATUS_SUCCESS	nvGRAPH operation was successful
NVGRAPH_STATUS_NOT_INITIALIZED	The nvGRAPH library was not initialized. This is usually caused by the lack of a prior call, an error in the CUDA Runtime API called by the nvGRAPH routine, or an error in the hardware setup. **To correct:** call **nvgraphCreate()** prior to the function call; and check that the hardware, an appropriate version of the driver, and the nvGRAPH library are correctly installed.
NVGRAPH_STATUS_ALLOC_FAILED	Resource allocation failed inside the nvGRAPH library. This is usually caused by a **cudaMalloc()** failure.
NVGRAPH_STATUS_INVALID_VALUE	An unsupported value or parameter was passed to the function **To correct:** ensure that all the parameters being passed have valid values.
NVGRAPH_STATUS_ARCH_MISMATCH	The function requires a feature absent from the device architecture. **To correct:** compile and run the application on a device with appropriate compute capability.
NVGRAPH_STATUS_MAPPING_ERROR	An access to GPU memory space failed.
NVGRAPH_STATUS_EXECUTION_FAILED	The GPU program failed to execute. This is often caused by a launch failure of the kernel on the GPU, which can be caused by multiple reasons.

	To correct: check that the hardware, an appropriate version of the driver, and the nvGRAPH library are correctly installed.
NVGRAPH_STATUS_INTERNAL_ERROR	An internal nvGRAPH operation failed. **To correct:** check that the hardware, an appropriate version of the driver, and the nvGRAPH library are correctly installed. Also, check that the memory passed as a parameter to the routine is not being deallocated prior to the routine's completion.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type is not supported by this function. This is usually caused by passing an invalid graph descriptor to the function.
NVGRAPH_STATUS_NOT_CONVERGED	An algorithm failed to converge. **To correct:** ensure that all the parameters being passed have valid values for this algorithm, increase the maximum number of iteration and/or the tolerance.

2.2. nvGRAPH graph topology types

Graphs toplogy types. Defines storage format. Some algorithms can work only with specific topology types, see algorithms descriptions for the list of supported topologies.

```
typedef enum
{
   NVGRAPH_CSR_32 = 0,
   NVGRAPH_CSC_32 = 1,
} nvgraphTopologyType_t;
```

Topology types

NVGRAPH_CSR_32	Compressed Sparse Rows format (row major format). Used in SrSPMV algorithm. Use nvgraphCSRTopologyX_t topology structure for this format.
NVGRAPH_CSC_32	Compressed Sparse Column format (column major format). Used in SSSP, WidestPath and Pagerank algorithms. Use nvgraphCSCTopologyX_t topology structure for this format.

2.3. nvGRAPH topology structure types

Graphs toplogy structures are used to set or retrieve topology data. User should use structure that corresponds to the chosen topology type.

nvgraphCSRTopology32I_t

Used for **NVGRAPH_CSR_32** topology type

```
struct nvgraphCSRTopology32I_st {
  int nvertices;
  int nedges;
  int *source_offsets;
  int *destination_indices;
};
typedef struct nvgraphCSRTopology32I_st *nvgraphCSRTopology32I_t;
```

Structure fields

nvertices	Number of vertices in the graph
nedges	Number of edges in the graph.
source_offsets	Array of size **nvertices+1**, where **i** element equals to the number of the first edge for this vertex in the list of all outgoing edges in the **destination_indices** array. Last element stores total number of edges
destination_indices	Array of size **nedges**, where each value designates destanation vertex for an edge.

nvgraphCSCTopology32I_t

Used for **NVGRAPH_CSC_32** topology type

```
struct nvgraphCSCTopology32I_st {
  int nvertices;
  int nedges;
  int *destination_offsets;
  int *source_indices;
};
typedef struct nvgraphCSCTopology32I_st *nvgraphCSCTopology32I_t;
```

Structure fields

nvertices	Number of vertices in the graph
nedges	Number of edges in the graph.
destination_offsets	Array of size **nvertices+1**, where **i** element equals to the number of the first edge for this vertex in the list of all incoming edges in the **source_indices** array. Last element stores total number of edges
source_indices	Array of size **nedges**, where each value designates source vertex for an edge.

2.4. Function nvgraphCreate()

```
nvgraphStatus_t
   nvgraphCreate(nvgraphHandle_t *handle);
```

Creates only an opaque handle, and allocates small data structures on the host. This handle is used in all of the nvGRAPH functions, so this function should be called first, before any other calls are made to the library.

Input/Output

handle	Pointer to a **nvgraphHandle_t** object

Return Values

NVGRAPH_STATUS_SUCCESS	nvGRAPH successfully created the handle.
NVGRAPH_STATUS_ALLOC_FAILED	The allocation of resources for the handle failed.
NVGRAPH_STATUS_INTERNAL_ERROR	An internal driver error was detected.

2.5. Function nvgraphDestroy()

```
nvgraphStatus_t
   nvgraphDestroy(nvgraphHandle_t handle);
```

Destroys handle created with **nvgraphCreate()**. This will automatically release any allocated memory objects created with this handle, for example any graphs and their vertices' and edges' data. Any consecutive usage of this handle will be invalid. Any calls to the nvGRAPH API after nvgraphDestroy() is called will be invalid and will return 'NVGRAPH_UNINITIALIZED' errors.

Input

handle	The **nvgraphHandle_t** object of the handle to be destroyed.

Return Values

NVGRAPH_STATUS_SUCCESS	nvGRAPH successfully destroyed the handle.
NVGRAPH_STATUS_INVALID_VALUE	The **handle** parameter is not a valid handle.

2.6. Function nvgraphCreateGraphDescr()

```
nvgraphStatus_t
   nvgraphCreateGraphDescr(nvgraphHandle_t handle, nvgraphGraphDescr_t *descrG);
```

Creates opaque handle for a graph structure. This handle is required for any operation on the graph.

Input

handle	nvGRAPH library handle

Input/Output

descrG	Pointer to the empty **nvgraphGraphDescr_t** structure object.

Return Values

NVGRAPH_STATUS_SUCCESS	Success
NVGRAPH_STATUS_INVALID_VALUE	Bad library handle is provided
NVGRAPH_STATUS_ALLOC_FAILED	Cannot allocate graph descriptor.

2.7. Function nvgraphDestroyGraphDescr()

```
nvgraphStatus_t
   nvgraphDestroyGraphDescr(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG);
```

Destroys graph handle created with nvgraphCreateGraphDescr(). This won't release memory allocated for this graph until library handle is destroyed.

Input

handle	nvGRAPH library handle
descrG	Graph descriptor to be released

Return Values

NVGRAPH_STATUS_SUCCESS	Successful release of the graph descriptor
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Graph is stored with unknown type of data
NVGRAPH_STATUS_INVALID_VALUE	Invalid library handle or graph descriptor handle

2.8. Function nvgraphSetGraphStructure()

```
nvgraphStatus_t
   nvgraphSetGraphStructure( nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void* topologyData, nvgraphTopologyType_t TType);
```

This call sets a graph topology for the given graph descriptor. Graph topology should be set only once. Users should choose one of the supported topologies, fill in the corresponding structure for the graph structure initialization and provide a pointer to this structure and choosed the topology type as parameters **topologyData** and **TType**. Typically graph structure data includes a number of vertices, number of edges and connectivity information based on the topology. Look at the description of the corresponding topology structures for details.

Input

handle	nvGRAPH library handle
topologyData	Pointer to a filled structure of one of the types **{nvgraphCSRTopology32I_t, nvgraphCSCTopology32I_t}**. The particular type to be used is defined by parameter TType.
TType	Graph topology type. This value should be equal to one of the possible values of the enum **nvgraphTopologyType_t**. This defines what data structure should be provided by the **topologyData** parameter.

Input/Output

descrG	Graph descriptor. Must not have topology previously defined.

Return Values

NVGRAPH_STATUS_SUCCESS	Success
NVGRAPH_STATUS_INVALID_VALUE	Invalid library handle, topology data structure pointer or topology values, or graph topology was already set
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Provided topology type is not supported
NVGRAPH_STATUS_INTERNAL_ERROR	Unknown internal error was caught

2.9. Function nvgraphGetGraphStructure()

```
nvgraphStatus_t
   nvgraphGetGraphStructure( nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void* topologyData, nvgraphTopologyType_t TType);
```

This function allows users to retrieve a given graph's topology information plus number of vertices and edges in the graph. Users must provide a graph descriptor and its topology type to get this information, as well as an empty topology structure, where this information will be stored.

Input

handle	nvGRAPH library handle
descrG	Graph descriptor. This graph should have its topology structure previously set.
TType	Graph topology type. This value should be equal to one of the possible values of the enum **nvgraphTopologyType_t** and equal to the real topology type of the graph. This defines what data structure type should be provided by the **topologyData** parameter.

Input/Output

topologyData	Pointer to a structure of one of the types **{nvgraphCSRTopology32I_t, nvgraphCSCTopology32I_t}**. The particular type to be

	used is defined by parameter **TType**. If call is successfull, this structure will be filled with topology information

Return Values

NVGRAPH_STATUS_SUCCESS	Success
NVGRAPH_STATUS_INVALID_VALUE	Invalid library handle, graph descriptor or topology for the graph is not set.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Unsupported topology or graph's topology doesn't match provided parameter.

2.10. Function nvgraphAllocateEdgeData()

```
nvgraphStatus_t
   nvgraphAllocateEdgeData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      size_t numsets, cudaDataType_t *settypes);
```

Allocates one or more storages for the data associated with graph edges. Number of allocated storages is specified by the numsets parameter. Types for each of the allocated storages should be provided in the array settypes of size numsets. Right now nvGRAPH graphs are limited to have data storages to have same type and same size - all elements of settypes array should be the same and all of those storages will have number of elements equal to the number of edges in the graph. Vertices data allocated with nvgraphAllocateVerticesData() function should have the same datatype as edge data. These storages could later be used in other functions by indices from **0** to **numsets-1**. This function could be called successfully only once for each graph.

Input

handle	nvGRAPH library handle
numsets	Number of datasets to allocate for the edges. Should be more than 0.
settypes	Array of size **numsets** that specifies types of allocated datasets. All values in this array should be the same and match graph's datasets data type, if exists.

Input/Output

descrG	Descriptor of the graph for which edge data is allocated. Should not have previously allocated edge data and have it's topology properly initialized.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Invalid function parameters, inconsistent types in the type array, types doesn't match graph's type or graph is not initialized for data allocation.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Types provided in parameter are not supported.

2.11. Function nvgraphSetEdgeData()

```
nvgraphStatus_t
   nvgraphSetEdgeData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void *edgeData, size_t setnum, nvgraphTopologyType_t TType);
```

Update a specific edge value set (weights) of the graph with the user's provided values.

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information and the edge set **setnum**
***edgeData**	Pointer to the data to load into the edge value set. This entry expects to read one value for each edge. Conversions are not supported so the user's type before the void* cast should be equivalent to the one specified in **nvgraphAllocateEdgeData**
setnum	The identifier of the set to update. This assumes that **setnum** is one of the the edge set allocated in the past using **nvgraphAllocateEdgeData**. Sets have 0-based index
TType	Type of topology. Conversions are not supported so this should match the graph structure.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_INTERNAL_ERROR	An internal operation failed.

2.12. Function nvgraphGetEdgeData()

```
nvgraphStatus_t
   nvgraphGetEdgeData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void *edgeData, size_t setnum, nvgraphTopologyType_t TType);
```

Downloads one dataset associated with graph edges using **setnum** index to the user memoryspace. **edgeData** could point to either host or device memoryspace. Size of the data transfer depends on the edges number of the graph and graph's data type.

Input

handle	nvGRAPH library handle.
descrG	Graph descriptor. Graph should contain at least one data set associated with it's vertices
setnum	Index of the source data set of the graph edge data. Value should be between 0 and **edge_dataset_number-1**

TType	Value should match graph's topology type

Output

edgeData	Pointer to the user memoryspace where edge data will be stored. Could be either host or device memory and have at least **number_of_edges*sizeof(graph_data_type)** bytes.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Incorrect function parameter, graph has no associated edge data sets or topology type doesn't match.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Graph datatype or topology type is not supported.
NVGRAPH_STATUS_INTERNAL_ERROR	Memory copy failed.

2.13. Function nvgraphAllocateVertexData()

```
nvgraphStatus_t
   nvgraphAllocateVertexData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      size_t numsets, cudaDataType_t *settypes);
```

Allocates one or more storages for the data associated with graph vertices. Number of allocated storages is specified by the numsets parameter. Types for each of the allocated storages should be provided in the array settypes of size numsets. Right now nvGRAPH graphs are limited to have data storages to have same type and same size - all elements of settypes array should be the same and all of those storages will have number of elements equal to the number of vertices in the graph. Edge data allocated with nvgraphAllocateEdgeData() function should have the same datatype as vertex data. These storages could later be used in other functions by indices from **0** to **numsets-1**. This function could be called successfully only once for each graph.

Input

handle	nvGRAPH library handle
numsets	Number of datasets to allocate for the vertices. Should be more than 0.
settypes	Array of size **numsets** that specifies types of allocated datasets. All values in this array should be the same and match graph's datasets data type, if exists.

Input/Output

descrG	Descriptor of the graph for which edge data is allocated. Should not have previously allocated edge data and have it's topology properly initialized.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.

NVGRAPH_STATUS_INVALID_VALUE	Invalid function parameters, inconsistent types in the type array, types doesn't match graph's type or graph is not initialized for data allocation.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Types provided in parameter are not supported.

2.14. Function nvgraphSetVertexData()

```
nvgraphStatus_t
   nvgraphSetVertexData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void *vertexData, size_t setnum, nvgraphTopologyType_t TType);
```

Update a specific vertex value set of the graph with the user's provided values.

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information and vertex set **setnum**.
***vertexData**	Pointer to the data to load into the vertex value set. This entry expects to read one value for each vertex. Conversions are not supported so the user's type before the void* cast should be equivalent to the one specified in **nvgraphAllocateVertexData**
setnum	The identifier of the set to update. This assumes that **setnum** is one of the the vertex set allocated in the past using **nvgraphAllocateVertexData**. Sets have 0-based index
TType	Type of topology. Conversions are not supported so this should match the graph structure.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_INTERNAL_ERROR	An internal operation failed.

2.15. Function nvgraphGetVertexData()

```
nvgraphStatus_t
   nvgraphGetVertexData(nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
      void *vertexData, size_t setnum, nvgraphTopologyType_t TType);
```

Downloads one dataset associated with graph vertices using **setnum** index to the user memoryspace. **vertexData** could point to either host or device memoryspace. Size of the data transfer depends on the vertex number of the graph and graph's data type.

Input

handle	nvGRAPH library handle.

descrG	Graph descriptor. Graph should contain at least one data set associated with it's vertices
setnum	Index of the source data set of the graph vertex data. Value should be between 0 and **vertex_dataset_number-1**
TType	Value should match graph's topology type

Output

edgeData	Pointer to the user memoryspace where vertex data will be stored. Could be either host or device memory and have at least **number_of_vertices*sizeof(graph_data_type)** bytes.

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Incorrect function parameter, graph has no associated vertex data sets or topology type doesn't match.
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	Graph datatype or topology type is not supported.
NVGRAPH_STATUS_INTERNAL_ERROR	Memory copy failed.

2.16. Function nvgraphExtractSubgraphByVertex()

```
nvgraphStatus_t
   nvgraphExtractSubgraphByVertex(nvgraphHandle_t handle,
      nvgraphGraphDescr_t descrG, nvgraphGraphDescr_t subdescrG,
      int *subvertices, size_t numvertices);
```

Create a new graph by extracting a subgraph given an array of vertices, consisting of row indices in the graph incidence matrix; array must be: (i) free of duplicates; (ii) sorted in ascending order; (iii) must consist of indices with values between 0 and **graph_nvertices-1**.

Input

handle	nvGRAPH handle of the source graph (original graph)
descrG	nvGRAPH descriptor of the source graph (original graph)
subvertices	array containing vertex indices (row indices in graph incidence matrix) of the subgraph to be extracted; array must be: (i) free of duplicates; (ii) sorted in ascending order; (iii) must consist of indices with values between 0 and **graph_nvertices-1**.
numvertices	the size of subvertices[] array. Should be more than 0 and less or equal to the number of graph's vertices.

Output

subdescrG	nvGRAPH graph descriptor of the target graph (subgraph)

Return Values

NVGRAPH_STATUS_SUCCESS	nvGRAPH target (subgraph) was created succesfully.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type of specified nvGRAPH is not supported.

2.17. Function nvgraphExtractSubgraphByEdge()

```
nvgraphStatus_t
   nvgraphExtractSubgraphByEdge(nvgraphHandle_t handle,
      nvgraphGraphDescr_t descrG, nvgraphGraphDescr_t subdescrG,
      int *subedges, size_t numedges);
```

Create a new graph by extracting a subgraph given an array of edges, consisting of indices in the col_ind[] array of the the graph incidence matrix CSR representation); the array of edges must be: (i) free of duplicates; (ii) sorted in ascending order; (iii) must consist of indices with values between 0 and **graph_nedges-1**.

Input

handle	nvGRAPH handle of the source graph (original graph)
descrG	nvGRAPH descriptor of the source graph (original graph)
subedges	array containing edge indices (indices in the col_ind[] array of the the graph incidence matrix CSR representation) of the subgraph to be extracted; array must be: (i) free of duplicates; (ii) sorted in ascending order; (iii) must consist of indices with values between 0 and **graph_nedges-1**
numedges	the size of subedges[] array, should be more than 0 and less or equal to number of graph's edges

Output

subdescrG	nvGRAPH graph descriptor of the target graph (subgraph)

Return Values

NVGRAPH_STATUS_SUCCESS	nvGRAPH target (subgraph) was created succesfully.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type of specified nvGRAPH is not supported.

2.18. Function nvgraphWidestPath()

```
nvgraphStatus_t
   nvgraphWidestPath(nvgraphHandle_t handle, const nvgraphGraphDescr_t descrG,
      const size_t weight_index, const int *source_vert,
      const size_t widest_path_index);
```

Find the widest path from the vertex at source_index to every other vertex; this problem is also known as 'the bottleneck path problem' or 'the maximum capacity path problem'.

If some vertices are unreachable, the widest path to those vertices is -∞. In limited-precision arithmetic, this corresponds to -FLT_MAX or -DBL_MAX depending on the value type of the set (CUDA_R_32F or CUDA_R_64F respectively).

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information in **NVGRAPH_CSC_32**, at least 1 edge set (the capacity) and 1 vertex set (to store the result).
weight_index	Index of the edge set for the weights.
***source_vert**	Index of the source, using 0-based indexes.

Output

widest_path_index	The values strored inside the vertex set at widest_path_index (VertexData[widest_path_index]) are the widest path values. VertexData[widest_path_index][i] is the length of the widest path between source_vert and vertex i. If vertex i is not reachable from source_vert, VertexData[widest_path_index][i] = -∞. Users can get a copy of the result using **nvgraphGetVertexData**

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type of at least one vetex or edge set is not supported.
NVGRAPH_STATUS_INTERNAL_ERROR	An internal operation failed.

2.19. Function nvgraphSssp()

```
nvgraphStatus_t
   nvgraphSssp(nvgraphHandle_t handle, const nvgraphGraphDescr_t descrG,
      const size_t weight_index, const int *source_vert,
      const size_t sssp_index);
```

The Single Source Shortest Path (SSSP) algorithm calculates the shortest path distance from a single vertex in the graph to all other vertices.

If some vertices are unreachable, the shortest path to those vertices is ∞. In limited-precision arithmetic, that corresponds to FLT_MAX or DBL_MAX depending on the value type of the set (CUDA_R_32F or CUDA_R_64F respectively).

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information in **NVGRAPH_CSC_32**, at least 1 edge set (distances) and 1 vertex set (the shortest path lengths).
weight_index	Index of the edge set for the weights. The default value is 0, meaning the first edge set.
***source_vert**	Index of the source, using 0-based indexes.

Output

sssp_index	The values stored inside the vertex set at sssp_index (VertexData[sssp_index]) are the shortest path values. VertexData[sssp_index][i] is the length of the shortest path between source_vert and vertex i. If vertex i is not reachable from source_vert, VertexData[sssp_index][i] = ∞. User can get a copy of the result using **nvgraphGetVertexData**

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type of at least one vetex or edge set is not supported.
NVGRAPH_STATUS_INTERNAL_ERROR	An internal operation failed.

2.20. Function nvgraphSrSpmv()

```
nvgraphStatus_t
   nvgraphSrSpmv(nvgraphHandle_t handle, const nvgraphGraphDescr_t descrG,
      const size_t weight_index, const void *alpha, const size_t x_index,
      const void *beta, const size_t y_index, const nvgraphSemiring_t SR);
```

The Semi-Ring Sparse Matrix Vector multiplication is an operation of the type y = α * A * x + β y. Where :

- A is a weighted graph seen as a compressed sparse matrix in CSR, x and y are vectors, α and β are scalars

- (*,+) is a set of two binary operators operating on real values and satisfying semi-ring properties.

In nvGRAPH all semi-rings operate on a set (R) with two binary operators: + and * that satisfies:

- (R, +) is associative, commutative with additive identity (additive_identity + a = a)

- (R, *) is associative with multiplicative identity (multiplicative_identity * a = a)

- Left and Right multiplication is distributive over addition

- Additive identity = multiplicative null operator (null_operator * a = a * null_operator = null_operator).

nvGRAPH's approach for sparse matrix vector multiplication on the GPU is based on the CSRMV merge-path algorithm from Duane Merill. It is designed to handle arbitrary sparsity patterns in an efficient way by offering a good workload balance. As a result, this operation delivers consistent good performance even for networks with a power-law distribution of connections.

nvGRAPH has pre-defined useful semi-ring for graphs in nvgraphSemiring_t, so the user can select them directly.

Semi rings

Semiring	Set	Plus	Times	Add_ident	Mult_ident
NVGRAPH_PLUS_TIMES_SR	R	+	*	0	1
NVGRAPH_MIN_PLUS_SR	R U {-∞,+∞}	min	+	∞	0
NVGRAPH_MAX_MIN_SR	R U {-∞,+∞}	max	min	-∞	+∞
NVGRAPH_OR_AND_SR	{0.0, 1,0}	OR	AND	0	1

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information in **NVGRAPH_CSR_32**, at least 1 edge set (weights) and 2 vertex sets (input vector and output vector).
weight_index	Index of the edge set for the weights.
***alpha**	Scalar used for multiplication
x_index	Index of the vertex set for used for multiplication
***beta**	Scalar used for multiplication. If beta is zero, the vertex set at y_index does not have to be a valid input.
y_index	(optional) Index of the vertex set for used for the addition.
SR	The semi-ring type **nvgraphSemiring_t** which can be **NVGRAPH_PLUS_TIMES_SR, NVGRAPH_MIN_PLUS_SR, NVGRAPH_MAX_MIN_SR, NVGRAPH_OR_AND_SR.**

Output

Values at **y_index**	The values stored inside the set at y_index (VertexData[y_index]) are the result of the operation. User can get a copy of the result using **nvgraphGetVertexData**

Return Values

NVGRAPH_STATUS_SUCCESS	Success.
NVGRAPH_STATUS_INVALID_VALUE	Bad parameter(s).
NVGRAPH_STATUS_TYPE_NOT_SUPPORTED	The type of at least one vetex or edge set is not supported.
NVGRAPH_STATUS_INTERNAL_ERROR	An internal operation failed.

2.21. Function nvgraphPagerank()

```
nvgraphStatus_t
   nvgraphPagerank(nvgraphHandle_t handle, const nvgraphGraphDescr_t descrG,
      const size_t weight_index, const void *alpha,
      const size_t bookmark_index,
      const int has_guess, const size_t pagerank_index,
      const float tolerance, const int max_iter);
```

Find the PageRank vertex values for a graph with a given transition matrix (Markov chain), a bookmark vector of dangling vertices, and the damping factor. The transition matrix is sub-stochastic (ie. each row sums to 0 or 1) and has to be provided in column major order (ie. in CSC, which is equivalent to the transposed of the sub-stochastic matrix in CSR). The bookmark vector flags vertices without outgoing edges (also called dangling vertices).

This is equivalent to an eigenvalue problem where we compute the dominant eigenpair. By construction, the maximum eigenvalue is 1, only the eigenvector is interesting. nvGRAPH computes an approximation of the Pagerank eigenvector using the power method. The number of iterations depends on the properties of the network itself; it increases when the tolerance descreases and/or alpha increases toward the limiting value of 1.

The user is free to use default values or to provide inputs for the initial guess, tolerance and maximum number of iterations.

Input

handle	nvGRAPH library handle.
descrG	nvGRAPH graph descriptor, should contain the connectivity information in **NVGRAPH_CSC_32**, at least 1 edge set and 2 vertex sets.
weight_index	Index of the edge set for the transition probability.
***alpha**	The damping factor **alpha** represents the probability to follow an outgoing edge, standard value is 0.85. Thus **1.0-alpha**is the probability to “teleport” to a random node. **alpha**should be greater than 0.0 and strictly lower than 1.0.
bookmark_index	Index of the vertex set for the bookmark of dangling nodes (**VertexData[bookmark_index][i] = 1.0** if i is a dangling node, 0.0 otherwise).
has_guess	This parameter is used to notify nvGRAPH if it should use a user-provided initial guess. 0 means the user doesn't

	have a guess, in this case nvGRAPH will use a uniform vector set to `1/V`. If the value is 1 nvGRAPH will read VertexData[pagerank_index] and use this as initial guess. The initial guess must not be the vector of 0s. Any value other than 1 or 0 is treated as an invalid value.
`pagerank_index`	(optional) Index of the vertex set for the initial guess if **`has_guess=1`**
`tolerance`	Set the tolerance the approximation, this parameter should be a small magnitude value. The lower the tolerance the better the approximation. If this value is `0.0f`, nvGRAPH will use the default value which is `1.0E-6`. Setting too small a tolerance (less than 1.0E-6 typically) can lead to non-convergence due to numerical roundoff. Usually 0.01 and 0.0001 are acceptable.
`max_iter`	The maximum number of iterations before an answer is returned. This can be used to limit the execution time and do an early exit before the solver reaches the convergence tolerance. If this value is lower or equal to 0 nvGRAPH will use the default value, which is 500.

Output

Values at **`pagerank_index`**	The values stored inside the vertex set at pagerank_index (VertexData[pagerank_index]) are the PageRank values. VertexData[pagerank_index][i] is the PageRank of vertex i. Users can get a copy of the result using **`nvgraphGetVertexData`**

Return Values

`NVGRAPH_STATUS_SUCCESS`	The Pagerank iteration reached the desired **`tolerance`** in less than **`max_iter`** iterations
`NVGRAPH_STATUS_NOT_CONVERGED`	The Pagerank iteration did not reach the desired **`tolerance`** after **`max_iter`** iterations
`NVGRAPH_STATUS_INVALID_VALUE`	Bad parameter(s).
`NVGRAPH_STATUS_TYPE_NOT_SUPPORTED`	The type of at least one vertex or edge set is not supported. Currently we support float and double type values.

2.22. Function nvgraphStatusGetString()

```
const char*
   nvgraphStatusGetString(nvgraphStatus_t status);
```

Gets string description for the nvGRAPH C API statuses.

Input

`status`	Status returned from one of the C API functions

Return Values

Pointer to the string with the text description of the C API status.

Chapter 3. NVGRAPH CODE EXAMPLES

This chapter provides simple examples.

3.1. nvGRAPH pagerank example

```
void check(nvgraphStatus_t status) {
    if (status != NVGRAPH_STATUS_SUCCESS) {
        printf("ERROR : %d\n",status);
        exit(0);
    }
}
int main(int argc, char **argv) {
    size_t  n = 6, nnz = 10, vert_sets = 2, edge_sets = 1;
    float alpha1 = 0.9f; void *alpha1_p = (void *) &alpha1;
    // nvgraph variables
    nvgraphHandle_t handle; nvgraphGraphDescr_t graph;
    nvgraphCSCTopology32I_t CSC_input;
    cudaDataType_t edge_dimT = CUDA_R_32F;
    cudaDataType_t* vertex_dimT;
    // Allocate host data
    float *pr_1 = (float*)malloc(n*sizeof(float));
    void **vertex_dim = (void**)malloc(vert_sets*sizeof(void*));
    vertex_dimT = (cudaDataType_t*)malloc(vert_sets*sizeof(cudaDataType_t));
    CSC_input = (nvgraphCSCTopology32I_t) malloc(sizeof(struct
 nvgraphCSCTopology32I_st));
    // Initialize host data
    float weights_h[] = {0.333333f, 0.5f, 0.333333f, 0.5f, 0.5f, 1.0f,
 0.333333f, 0.5f, 0.5f, 0.5f};
    int destination_offsets_h[] = {0, 1, 3, 4, 6, 8, 10};
    int source_indices_h[] = {2, 0, 2, 0, 4, 5, 2, 3, 3, 4};
    float bookmark_h[] = {0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f};
    vertex_dim[0] = (void*)bookmark_h; vertex_dim[1]= (void*)pr_1;
    vertex_dimT[0] = CUDA_R_32F; vertex_dimT[1]= CUDA_R_32F, vertex_dimT[2]=
 CUDA_R_32F;
    // Starting nvgraph
    check(nvgraphCreate (&handle));
    check(nvgraphCreateGraphDescr (handle, &graph));
    CSC_input->nvertices = n; CSC_input->nedges = nnz;
    CSC_input->destination_offsets = destination_offsets_h;
    CSC_input->source_indices = source_indices_h;
    // Set graph connectivity and properties (tranfers)
    check(nvgraphSetGraphStructure(handle, graph, (void*)CSC_input,
 NVGRAPH_CSC_32));
    check(nvgraphAllocateVertexData(handle, graph, vert_sets, vertex_dimT));
    check(nvgraphAllocateEdgeData  (handle, graph, edge_sets, &edge_dimT));
    for (int i = 0; i < 2; ++i)
        check(nvgraphSetVertexData(handle, graph, vertex_dim[i], i,
 NVGRAPH_CSC_32));
    check(nvgraphSetEdgeData(handle, graph, (void*)weights_h, 0,
 NVGRAPH_CSC_32));

    check(nvgraphPagerank(handle, graph, 0, alpha1_p, 0, 0, 1, 0.0f, 0));
    // Get result
    check(nvgraphGetVertexData(handle, graph, vertex_dim[1], 1,
 NVGRAPH_CSC_32));
    check(nvgraphDestroyGraphDescr(handle, graph));
    check(nvgraphDestroy(handle));
    free(pr_1); free(vertex_dim); free(vertex_dimT);
    free(CSC_input);
    return 0;
}
```

3.2. nvGRAPH SSSP example

```
void check(nvgraphStatus_t status) {
    if (status != NVGRAPH_STATUS_SUCCESS) {
        printf("ERROR : %d\n",status);
        exit(0);
    }
}
int main(int argc, char **argv) {
    const size_t  n = 6, nnz = 10, vertex_numsets = 1, edge_numsets = 1;
    float *sssp_1_h;
    void** vertex_dim;
    // nvgraph variables
    nvgraphStatus_t status; nvgraphHandle_t handle;
    nvgraphGraphDescr_t graph;
    nvgraphCSCTopology32I_t CSC_input;
    cudaDataType_t edge_dimT = CUDA_R_32F;
    cudaDataType_t* vertex_dimT;
    // Init host data
    sssp_1_h = (float*)malloc(n*sizeof(float));
    vertex_dim  = (void**)malloc(vertex_numsets*sizeof(void*));
    vertex_dimT =
 (cudaDataType_t*)malloc(vertex_numsets*sizeof(cudaDataType_t));
    CSC_input = (nvgraphCSCTopology32I_t) malloc(sizeof(struct
 nvgraphCSCTopology32I_st));
    vertex_dim[0]= (void*)sssp_1_h; vertex_dimT[0] = CUDA_R_32F;
    float weights_h[] = {0.333333, 0.5, 0.333333, 0.5, 0.5, 1.0, 0.333333, 0.5,
 0.5, 0.5};
    int destination_offsets_h[] = {0, 1, 3, 4, 6, 8, 10};
    int source_indices_h[] = {2, 0, 2, 0, 4, 5, 2, 3, 3, 4};
    check(nvgraphCreate(&handle));
    check(nvgraphCreateGraphDescr (handle, &graph));
    CSC_input->nvertices = n; CSC_input->nedges = nnz;
    CSC_input->destination_offsets = destination_offsets_h;
    CSC_input->source_indices = source_indices_h;
    // Set graph connectivity and properties (tranfers)
    check(nvgraphSetGraphStructure(handle, graph, (void*)CSC_input,
 NVGRAPH_CSC_32));
    check(nvgraphAllocateVertexData(handle, graph, vertex_numsets,
 vertex_dimT));
    check(nvgraphAllocateEdgeData  (handle, graph, edge_numsets, &edge_dimT));
    check(nvgraphSetEdgeData(handle, graph, (void*)weights_h, 0,
 NVGRAPH_CSC_32));
    // Solve
    int source_vert = 0;
    check(nvgraphSssp(handle, graph, 0,  &source_vert, 0));
    // Get and print result
    check(nvgraphGetVertexData(handle, graph, (void*)sssp_1_h, 0,
 NVGRAPH_CSC_32));
    //Clean
    free(sssp_1_h); free(vertex_dim);
    free(vertex_dimT); free(CSC_input);
    check(nvgraphDestroyGraphDescr(handle, graph));
    check(nvgraphDestroy(handle));
    return 0;
}
```

3.3. nvGRAPH Semi-Ring SPMV example

```
void check(nvgraphStatus_t status) {
    if (status != NVGRAPH_STATUS_SUCCESS) {
        printf("ERROR : %d\n",status);
        exit(0);
    }
}
int main(int argc, char **argv) {
    size_t  n = 5, nnz = 10, vertex_numsets = 2, edge_numsets = 1;
    float alpha = 1.0, beta = 0.0;
    void *alpha_p = (void *)&alpha, *beta_p = (void *)&beta;
    void** vertex_dim;
    cudaDataType_t edge_dimT = CUDA_R_32F;
    cudaDataType_t* vertex_dimT;
    // nvgraph variables
    nvgraphStatus_t status; nvgraphHandle_t handle;
    nvgraphGraphDescr_t graph;
    nvgraphCSRTopology32I_t CSR_input;
    // Init host data
    vertex_dim  = (void**)malloc(vertex_numsets*sizeof(void*));
    vertex_dimT =
 (cudaDataType_t*)malloc(vertex_numsets*sizeof(cudaDataType_t));
    CSR_input = (nvgraphCSRTopology32I_t) malloc(sizeof(struct
 nvgraphCSRTopology32I_st));
    float x_h[] = {1.1f, 2.2f, 3.3f, 4.4f, 5.5f};
    float y_h[] = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f};
    vertex_dim[0]= (void*)x_h; vertex_dim[1]= (void*)y_h;
    vertex_dimT[0] = CUDA_R_32F; vertex_dimT[1]= CUDA_R_32F;
    float weights_h[] = {1.0f, 4.0f, 2.0f, 3.0f, 5.0f, 7.0f, 8.0f, 9.0f, 6.0f,
 1.5f};
    int source_offsets_h[] = {0, 2, 4, 7, 9, 10};
    int destination_indices_h[] = {0, 1, 1, 2, 0, 3, 4, 2, 4, 2};
    check(nvgraphCreate(&handle));
    check(nvgraphCreateGraphDescr(handle, &graph));
    CSR_input->nvertices = n; CSR_input->nedges = nnz;
    CSR_input->source_offsets = source_offsets_h;
    CSR_input->destination_indices = destination_indices_h;
    // Set graph connectivity and properties (tranfers)
    check(nvgraphSetGraphStructure(handle, graph, (void*)CSR_input,
 NVGRAPH_CSR_32));
    check(nvgraphAllocateVertexData(handle, graph, vertex_numsets,
 vertex_dimT));
    for (int i = 0; i < vertex_numsets; ++i)
       check(nvgraphSetVertexData(handle, graph, vertex_dim[i], i,
 NVGRAPH_CSR_32));
    check(nvgraphAllocateEdgeData  (handle, graph, edge_numsets, &edge_dimT));
    check(nvgraphSetEdgeData(handle, graph, (void*)weights_h, 0,
 NVGRAPH_CSR_32));
    // Solve
    check(nvgraphSrSpmv(handle, graph, 0, alpha_p, 0, beta_p, 1,
 NVGRAPH_PLUS_TIMES_SR));
    //Get result
    check(nvgraphGetVertexData(handle, graph, (void*)y_h, 1, NVGRAPH_CSR_32));
    //Clean
    check(nvgraphDestroyGraphDescr(handle, graph));
    check(nvgraphDestroy(handle));
    free(vertex_dim); free(vertex_dimT); free(CSR_input);
    return 0;
}
```