Simulating Large Topologies in ns-3 using BRITE and CUDA Driven Global Routing

Brian Paul Swenson, George F. Riley
School of Electrical and Computing Engineering
Georgia Institute of Technology, Atlanta, GA, USA
{bpswenson, riley}@gatech.edu

ABSTRACT

We present two new modules for ns-3. The first is a BRITE module that allows ns-3 users to take advantage of the topology generation features of the BRITE topology generator. This module allows users to quickly create highly customizable, large scale topologies for simulation within ns-3. The second module is a new global routing module specifically designed to work with large ns-3 network topologies. This module leverages the parallel processing abilities of GPUs using CUDA. Using this new routing module on large topologies, we measured speedups of over three orders of magnitude compared to ns-3's current global routing protocol when performing global route computations.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: CUDA; I.6 [**Simulation and Modelling**]: General

General Terms

Simulation, Algorithm, Design, Performance

Keywords

ns-3, CUDA, BRITE, GPGPU

1. INTRODUCTION

Due to the complexities of the internet, the use of simulation tools is essential for examining the effectiveness of new protocols and the practicality of new internet applications. In many cases, in order to get a full understanding of the impact of the new protocol or application, the simulation needs to be performed on a large-scale topology with many competing sources of traffic. However, constructing large-scale topologies that exhibit the same characteristics of real networks is not a trivial task. Furthermore, once the topology is constructed, the generation of the routing tables needed to direct packets across the network can be extremely time intensive. The goal of this work is provide ns-3 users with the means to be able to generate large-scale topologies which are representative of real networks and simulate them in a time efficient manner.

This paper presents two new modules for use within ns-3. The first module is an ns-3 interface to the BRITE topology generator library. BRITE allows users to quickly create customizable, large-scale topologies for simulation. Our module takes the topology generated by BRITE and creates an identical representation in ns-3. Our module also provides helper methods which makes it easy for the user to configure the nodes in the generated topology. Futhermore, the BRITE module also works with MPI, allowing even larger networks by spreading the processing burden across multiple CPUs. The second module is a new global routing module that has been specifically designed to work with large-scale topologies. This module takes advantage of the massive parallel execution architecture of GPUs using NVIDIA's CUDA programming model. Using our new global routing module, we are able to perform global routing tasks at a rate much faster than is possible with the current global routing protocol in ns-3.

The remainder of the paper is organized as follows. In section 2 we discuss our new BRITE module. Section 3 describes the CUDA programming environment, the Floyd-Warshall algorithm and related work in this area. Section 4 describes our implementation in ns-3. In section 5 we present our experiments and results. Section 6 outlines next steps and future work. Finally, in section 7 we conclude our work.

2. BRITE

BRITE[10], the Boston University Representative Internet Topology Generator, is a common tool for generating realistic internet topologies for simulation. BRITE is a widely used topology generator that is used in many simulation applications such as ns-2, GTNetS, SSF and OmNet++. Topology construction in BRITE is highly customizable and is controlled by user provided configuration scripts. Included in BRITE is a front-end GUI to ease the process of creating these configuration files. Using BRITE, a wide variety of different topologies can be generated including scale free (power law) networks which are commonly seen within the internet[2].

BRITE uses two main algorithms for placing nodes in generated topologies, the Waxman[15] model and the Barbasi-Albert[2] model. The Waxman model is based originally on the Erdös-Renyi[6] random graph model. The Waxman

WNS3 2013, March 05-07

DOI 10.4108/icst.simutools.2013.251732

model differs from the Erdös-Renyi random graph model in that all generated nodes are placed on a plane and connectivity between nodes is based on euclidean distance. Specifically, the probability that two nodes, x and y, are connected is given by the formula:

$$P(x, y) = \alpha e^{-d/\beta L} \quad (1)$$

where d is the distance from node x to node y, L is the euclidean diameter of the network, and α and β are parameters.

The other model used by BRITE is the Barbasi-Albert model. This model can be used to create scale-free networks. This model uses incremental growth and preferential attachment to create topologies which conform with a power law. In this model, nodes are connected to the topology incrementally. As each node is added, it is more likely to connect to nodes that are highly connected. Specifically, the probability that node x, which is joining the network, is connected to node y, a node already in the network, is given by:

$$P(x, y) = \frac{d_y}{\sum_{k \in V} d_k} \quad (2)$$

where d_i is the degree of node i and V is the set of nodes already in the network. Therefore the denominator is the sum of the outgoing edges of all nodes already in the network.

There are three major types of topologies available in BRITE: Router, Autonomous System (AS), and Hierarchical which is a combination of AS and router. For the purposes of ns-3 simulation, the most useful topologies are likely to be Router and Hierarchical because both of these result in a graph of routers. Router level topologies are generated using either the Waxman model or the Barbasi-Albert model. Each model has different parameters that effect topology creation and these are specified in the configuration file. For flat router topologies, all nodes are considered to be in the same AS.

Hierarchical topologies contain two levels. First there is a top level AS topology, which can be constructed using either the AS-Waxman model or the AS-Barbasi-Albert model. These models are equivalent to the router level versions of the models. After the top level is constructed, for each node in the AS graph, a router level topology is constructed. These router level topologies can be created using a different model and different parameters than used in the AS model. BRITE then connects the different router level topologies using the original AS model as a guide. The algorithms used to make these connections were borrowed from the GT-ITM topology generator[4]. At this point the topology is fully connected and BRITE moves on to assigning bandwidths.

Once the topology is generated, the next step BRITE performs is to assign bandwidths to the links. BRITE offers four different possible distributions for assigning bandwidths: Constant, Uniform, Exponential and Heavy-tailed. The parameters used to control these distributions, BWdist, BWmin and BWmax, are specified in the BRITE configuration file. It should be noted that BRITE treats bandwidth values as unit-less. How ns-3 handles these values will be explained in in the next section.

2.1 BRITE integration with ns-3

The ns-3 interaction with BRITE occurs through the BriteTopologyHelper class. The construction of the BRITE topology is controlled by a user provided configuration file. Therefore the constructor for the helper takes as a parameter a string specifying the path to a BRITE configuration file. Users can also optionally specify a BRITE seed file to be used in construction of the topology. During topology construction, BRITE uses a pseudo-random number generator and therefore seeds are needed to create different topologies with similar characteristics.

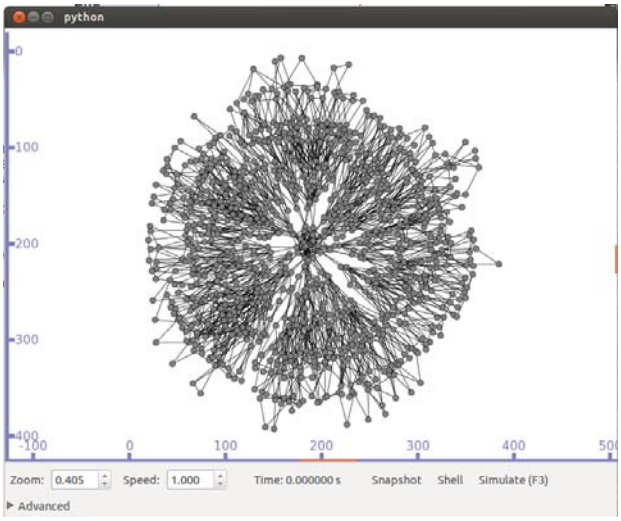


Figure 1: Sample BRITE hierarchical topology. The top level used the AS-Barbasi-Albert model and contained 20 nodes. The router level was configured using the Waxman model. Each AS contains 50 nodes for a total of 1000 (20 * 50) nodes.

An alternative to providing a seed file is to use ns-3's uniform distribution random number generator to automatically generate the seed values required by BRITE. An instance of this class is instantiated within the topology helper. The helper class provides an accessor method called AssignStreams to set the stream value of the generator. If a BRITE seed file is not passed in via the constructor, the helper will use the random number generator to generate the seed values. When the BRITE library executes, it automatically generates a file named last_seed_file that contains the seeds used in the generation of the last topology. Therefore, if an experimenter wanted to save the seed values in order to regenerate the same topology again, they could save a copy of this file prior to another invocation of the BRITE library.

Once an object of type BriteTopologyHelper has been created and configured, the next step is to actually create the new topology. To facilitate this, a method called BuildBriteTopology is provided by the helper class. There are two versions of this method. Both versions accept as a parameter an InternetStackHelper instance. This stack is installed on all nodes created in the BRITE topology.

The BRITE module has also been designed to optionally work with MPI. One version of BuildBriteTopology accepts an unsigned int representing the system count. Each AS in the generated topology is assigned a system number based on a modulus divide. Then, when the helper is generating the ns-3 topology, ns-3 nodes are created on the system corresponding to the system number of the AS they belong to. The system number given to an AS can be found by

using the GetSystemNumberForAs method provided by the helper.

The BriteTopologyHelper uses the BRITE library to generate a topology and then uses this BRITE topology to generate an equivalent version in ns-3. Every node in the BRITE topology has a matching node in the ns-3 topology. Every link in the BRITE topology is represented by a point to point link in the ns-3 topology. When assigning bandwidths to the point to point links in the topology, the helper assumes the values provided by BRITE are specified in Mbps. Therefore, when creating a BRITE configuration script, it is important to specify the bandwidth parameters in Mbps.

In the process of topology generation, BRITE classifies nodes depending on their connectivity in the network. Router level nodes are classified into one of the following categories: Leaf, Border, Backbone, Stub and None. Typically when working with BRITE generated topologies the most useful of these are the leaf nodes. The leaf nodes can be used to attach any other ns-3 topology, including other ns-3 BRITE topologies. To provide access to the leaf nodes, the BriteTopologyHelper class provides a GetLeafNodeForAs method which accepts as parameters an AS number and a leaf node index. Access to all of the nodes in the topology is available using the GetNodeForAs method. See the example code below for an example.

Once the ns-3 version of the topology has been generated, the next step is to assign IP addresses to the nodes in the newly created topology. This is accomplished using the method AssignIpv4Addresses or AssignIpv6Addresses, provided by the helper. When assigning IP addresses to the interfaces in the topology, each point to point link is treated as a separate network. Therefore it is important, especially for Ipv4, to set the size of the subnetwork to an appropriate size. Otherwise a large portion of the available address space will be wasted.

2.2 Example using BRITE

Here we will show a small code sample using the BriteTopologyHelper class.

```
// Add required header files
#include "ns3/brite-module.h"

// Invoke the BriteTopologyHelper and pass
// in a BRITE configuration file.  This will
// use BRITE to build a graph from which we
// can build the ns-3 topology
BriteTopologyHelper bth (confFile);

InternetStackHelper stack;

// The following builds the BRITE topology
// and then creates a ns-3 representation.
// The stack helper is passed in so all newly
// created nodes can have a stack installed
// prior to links being created
bth.BuildBriteTopology (stack);

Ipv4AddressHelper address;

// A small subnet is used because all of the
// point to point links created in the topology
// are assigned IP addresses as a separate network.
// Therefore by using a small subnetwork we don't
// waste any of the address space
address.SetBase ("10.0.0.0", "255.255.255.252");
bth.AssignIpv4Addresses (address);

// Iterate through all of the nodes in the
// BRITE generated topology
for (uint32_t i = 0; i < bth.GetNAs (); ++i)
  for (uint32_t j=0; j < bth.GetNNodesForAs(i); ++j)
  {
    //perform action on all nodes in topology
    Ptr<Node> node = bth.GetNodeForAs (i, j);
  }

// Iterate through all of the leaf nodes in the
// BRITE generated topology
for (uint32_t i = 0; i < bth.GetNAs (); ++i)
  for (uint32_t j = 0; j <
   bth.GetNLeafNodesForAs(i); ++j)
  {
    //perform action on all leafs nodes in topology
    Ptr<Node> leaf = bth.GetLeafNodeForAs (i, j);
  }
```

2.3 Status of BRITE in ns-3

The ns-3 BRITE interface implementation has completed peer review and will be included in the ns-3 release 3.16. The BRITE module is an optional module that can be enabled using waf configure –with-brite=PathToBriteLibrary. One of the main problems we ran into when testing with BRITE generated topologies is that it took an extremely long time to populate routing tables when the number of nodes in the topology started to become large. We present actual timings for a variety of topology sizes in the Experiments section. These large delays led us to look for alternative means to generate this route information.

3. CUDA

CUDA[11] (Compute Unified Device Architecture) is a general purpose parallel computing platform and programming model that can use the parallel execution abilities of NVIDIA GPUs (Graphics Processing Units). When programming with CUDA, the GPU is seen as a coprocessor with the ability to execute a large number of threads in parallel. To execute code on the GPU, the host computer uploads the data and the compiled CUDA program, called a kernel, to the GPU. The kernel is then invoked and the CPU can either continue processing other tasks or wait until kernel execution is complete.

CUDA threads are executed on the device in warps which are scheduled onto one of the GPU's streaming multiprocessors (SM). Currently in CUDA there are 32 threads per warp. The threads in a warp operate in a SIMD (Single Instruction Multiple Data) format. Therefore it is extremely important to avoid divergent code paths in kernels because both paths will have to be executed serially. Threads are also further grouped into blocks, as shown in figure 2. The number of threads per block is chosen by the programmer, although there is a maximum value which depends on the particular graphics card. The size should also be a multiple of the warp size. Common maximum threads per block values are 512 and 1024. A group of blocks is known as a grid.

All threads within a block are guaranteed to execute on the same streaming multiprocessor. Threads within the same block have access to a fast shared memory pool and can have the ability to synchronize at specific points. Dif-

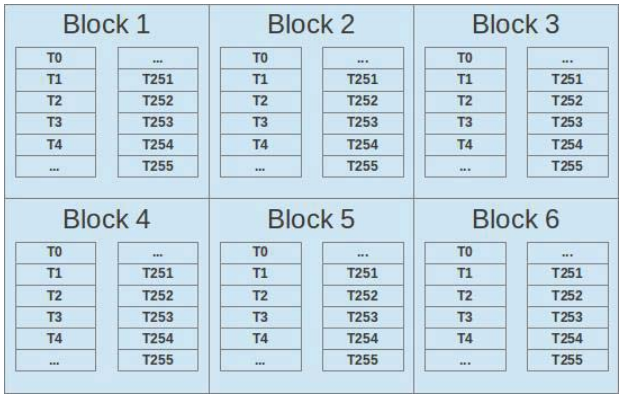


Figure 2: Here is a grid of thread blocks. Each block is configured with 256 threads. The kernel code associated with these blocks will be executed 1,536 (6 x 256) times.

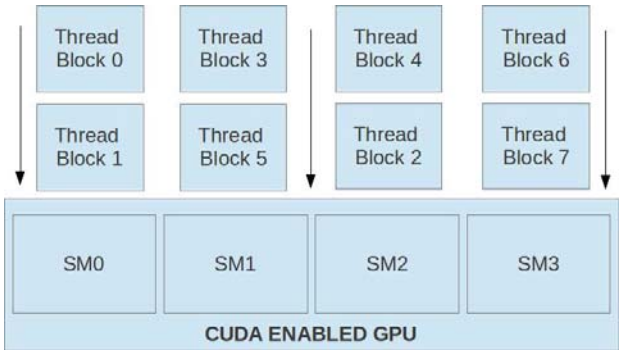


Figure 3: All threads within a block are guaranteed to run on a single streaming multiprocessor which allows synchronization among threads within a block. Blocks themselves however may be run in any order and there is no synchronization among threads in different thread blocks within the kernel

ferent thread blocks however, execute independently. They may be scheduled in any order, in series, or in parallel as shown in figure 3. Therefore, synchronization is not possible within the kernel between threads operating in different thread blocks. CUDA does however provide a way to synchronize between blocks outside of the kernel. The CudaDeviceSynchronize method halts CPU execution until all blocks submitted for execution finish.

3.1 Floyd-Warshall and CUDA

The Floyd-Warshall algorithm is a dynamic programming algorithm that can be used to solve the all-pairs shortest path problem on a graph. The algorithm runs in $\theta(N^3)$ time where N is the number of nodes in the graph to be solved. The algorithm operates on a NxN matrix. The initial matrix is an adjacency matrix where cell(i,j) specifies the weight of the edge going from node i to node j. If nodes are not directly connected, the initial value for the cell is infinity. The algorithm then proceeds as follows:

Basic Floyd-Warshall

```
1: for k = 1 to N do
2:     for i = 1 to N do
3:         for j = 1 to N do
4:             if W[i][k] + W[k][j] < W[i][j] then
5:                 W[i][j] = W[i][k] + W[k][j]
6:             end if
7:         end for
8:     end for
9: end for
```

In general terms the algorithm works as following. At each step, k, the distance from i to j, for all pairs in the matrix, is checked to see if it can be improved by using vertex k as an intermediate. A full description of the algorithm and a proof of its correctness was described by Corman[5].

This algorithm is an attractive fit for CUDA because all of the comparisons along i and j for a given k are completely independent and can be performed in parallel. Furthermore, the algorithm contains no divergent paths, one if and no else, so it maps extremely well into CUDA's SIMD execution architecture.

The Floyd-Warshall algorithm provides the distance on the shortest path between all pairs. However for routing we are interested in the actual path itself. Thankfully with a slight modification, the Floyd-Warshall algorithm can provide enough information to generate the shortest path:

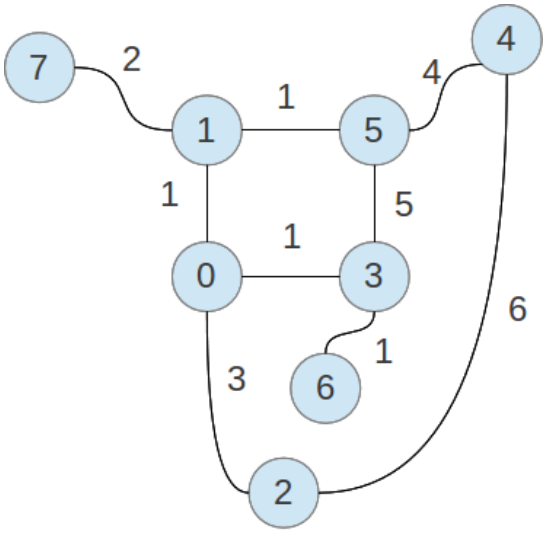
Floyd-Warshall with Path

```
1: for k = 1 to N do
2:     for i = 1 to N do
3:         for j = 1 to N do
4:             if W[i][k] + W[k][j] < W[i][j] then
5:                 W[i][j] = W[i][k] + W[k][j]
6:                 next[i][j] = k
7:             end if
8:         end for
9:     end for
10: end for
```

Here variable *next* is another NxN matrix. The values for *next* are initialized to j for all cell(i,j). Following the completion of the algorithm, *next* will contain the highest index vertex, k, that i must travel through along the shortest path to j. The *next* matrix also contains the highest index vertex, k_1, that i must travel through along the shortest path to k and so on. Using this information, it is possible to determine the next hop for i along the shortest path to j. Figure 4 provides an example of this process. Following the completion of the Floyd-Warshall algorithm (b), for node 5 to travel to node 6 it must travel through node 3, which is a neighbor of node 5, but it is not along the shortest path. The *next* matrix is examined to determine the shortest path from node 5 to node 3 and it is found that it is through node 1. The matrix is again examined to determine the shortest path from node 5 to node 1 and it is found that it is through node 1. Node 1 is now saved as the next hop on the shortest path from node 5 to node 6.

3.2 Related Work

Harnish and Narayanan[7] examined the performance of multiple graph algorithms on the GPU. They first examined a Breadth First Search (BFS) implementation. For this they used a compact adjacency list instead of an adjacency matrix. For sparse graphs, which they were using to test, this significantly reduced the amount of data needed to be stored and transferred to the GPU. The compact adjacency list used two arrays, one for nodes and one for edges that was sorted by edge source node. Each entry in the node array contained a pointer to its first edge in the edge



Initial Adjacency Graph

	0	1	2	3	4	5	6	7
0	0	1	3	1	-	-	-	-
1	1	0	-	-	-	1	-	2
2	3	-	0	-	6	-	-	-
3	1	-	-	0	-	5	1	-
4	-	-	6	-	0	4	-	-
5	-	1	-	5	4	0	-	-
6	-	-	-	1	-	-	0	-
7	-	2	-	-	-	-	-	0

Initial Next Graph

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	1	2	3	4	5	6	7
3	0	1	2	3	4	5	6	7
4	0	1	2	3	4	5	6	7
5	0	1	2	3	4	5	6	7
6	0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6	7

(a)

Shortest Path Distance

	0	1	2	3	4	5	6	7
0	0	1	3	1	6	2	2	3
1	1	0	4	2	5	1	3	2
2	3	4	0	4	6	5	5	6
3	1	2	4	0	7	3	1	4
4	6	5	6	7	0	4	8	7
5	2	1	5	3	4	0	4	3
6	2	3	5	1	8	4	0	5
7	3	2	6	4	7	3	5	0

Next Graph After FW

	0	1	2	3	4	5	6	7
0	0	1	2	3	5	1	3	1
1	0	1	0	0	5	5	3	7
2	0	0	2	0	4	1	3	1
3	0	0	0	3	5	1	6	1
4	5	5	2	5	4	5	5	5
5	1	1	1	1	4	5	3	1
6	3	3	3	3	5	3	6	3
7	1	1	1	1	5	1	3	7

(b)

Next Hop Node

	0	1	2	3	4	5	6	7
0	0	1	2	3	1	1	3	1
1	0	1	0	0	5	5	0	7
2	0	0	2	0	4	0	0	0
3	0	0	0	3	0	0	6	0
4	5	5	2	5	4	5	5	5
5	1	1	1	1	4	5	1	1
6	3	3	3	3	3	3	6	3
7	1	1	1	1	1	1	1	7

(c)

Figure 4: An example showing how FW can be used to construct shortest paths. (a) Initialization. (b) The results in the original adjacency matrix and the next matrix following the FW algorithm. (c) The final routes computed.

array. They also constructed a Single Source Shortest Path (SSSP) kernel. This algorithm worked with graphs with positive weights and found the combined weight of the shortest path from a source to every other node in the graph. Finally they created two All Pairs Shortest Path (APSP) kernels. For the first they used the Floyd-Warshall Algorithm with an adjacency matrix and for the second they repeated their SSSP kernel N times using the compact adjacency list. They found that the multi-SSSP kernel scaled better and also performed better than their Floyd-Warshall kernel. It should be noted that none of their CUDA implementations actually produced an actual path, they just calculated either the minimum number of edges (BFS), or the minimum path weight (SSSP and APSP) between nodes. Further work would need to be done to efficiently produce the actual paths using CUDA.

Katz and Kider[8] developed their own CUDA version of APSP using the Floyd-Warshall algorithm. Their approach extended previous work by Venkataraman[14] who produced a blocked, cache efficient adaptation of APSP for CPUs. Both implementations essentially break the initial adjacency matrix into submatrices which can be processed in a distributed, parallel manner. This approach allows for larger topologies than would normally fit into GPU memory. It also allows the ability to take advantage of multiple-GPU setups. With their GPU implementation, Katz and Kider reported a speedup of 6.5x the previous work done by Harnish and Narayanan. Again no actual path was calculated in their work; they offered that as potential future work.

Further optimizations to the APSP CUDA model presented by Katz were offered by Lund and Smith[9]. They performed strength reduction to reduce the number of expensive instructions. Next they reduced the amount of shared memory required by each thread block. They did this by making more efficient use of onboard registers and by staging the load of submatrices to reduce the number that needed to be loaded into shared memory simultaneously. These changes resulted in a speedup of approximately 5.2x over work performed by Katz.

Our implementation of APSP using CUDA most closely resembles the Floyd-Warshall APSP kernel present by Harnish and Narayanan. There were a number of reasons why we chose to go this route. First the procedure was straight forward to implement and modify to include generation of the actual routes detected. While faster results were obtained using alternative approaches[3][9][8], all of these approaches made extensive use of scarce GPU resources, such as registers and thread block shared memory, and do not consider the additional overhead of storing and manipulating a large NxN *next* matrix, which is essential to be able to generate the actual routes. This is important because it has been shown that even small changes in kernel design can have extremely large effects on performance[13].

4. NS-3 CUDA ROUTING

Our CUDA global routing module, like our BRITE module, is designed to be an optional ns-3 module. The code that runs the actual CUDA kernel performing the Floyd-Warshall algorithm has been placed into a shared library that is optionally linked into ns-3 with waf –with-fwrouter configure. Along with the shared library we have created a new routing module along with a helper class.

4.1 Generation of Routes

The first step performed with our new routing process is to examine the ns-3 node topology and generate the information required for routing. Three main pieces of information are gathered in this process. First is an IP address to node number map. In order to reduce the amount of data that needs to be transferred to the GPU device and to speed up the route computation, we compute routes node to node as opposed to net device to net device. Because of this, we need an efficient way to translate a net device's IP address to a node number. This map is generated once, stores all of the IP address in the topology and is made available to all nodes participating in the global routing.

The second piece of information generated in this process is the global adjacency matrix. This NxN matrix is used as the input to the Floyd-Warshall algorithm. To gather this information, we cycle through the net devices on a node and compile a list of adjacent net devices. The code we used to gather adjacent net devices for a net device is the same as the code performing the same task in the Nix Vector routing module. Once this information is obtained, we fill out that node's corresponding row in the adjacency matrix. Currently we are assigning each link with a weight of one; however, that is not a requirement. The way the kernel is programmed, it will work with any integer weight.

Finally, for each node, we create and store a partial routing table containing only its immediate neighbors. The information is stored in a map indexed by the neighbor's node number and contains the interface number and the remote IP address of the adjacent device. This map allows us to quickly obtain the next hop routing information we need when it becomes time to route a packet. At this point all of the necessary information has been gathered from the ns-3 topology and control is passed to the shared library where the routes are computed.

Once control has been passed to the shared library, the first thing it does is allocate memory on the GPU for the adjacency matrix and the *next* matrix. Again, these are both NxN integer matrices. Next the adjacency matrix, generated from the ns-3 topology, is copied to the GPU device and the *next* matrix is initialized. Since the *next* matrix has a large number of values that must be initialized, this is done via a separate CUDA kernel program. This allows this process to occur very quickly since each value can be initialized by an individual CUDA thread.

At this point, processing of the Floyd-Warshall algorithm can begin. As mentioned previously, during each step of the Floyd-Warshall algorithm all of the values in the working adjacency matrix need to be checked to see if a shorter path is found from vertex i to vertex j using vertex k as an intermediate. For a given k, all of the paths in the working matrix can be updated in parallel but synchronization needs to occur after each value for k. Since it is quite possible the number of paths in the topology will exceed the maximum number of threads per block, we cannot perform the entire algorithm with one kernel call. Again this is because it is not possible to synchronize between blocks within a kernel. Therefore our implementation makes a separate kernel call for each value of k and synchronizes at the CPU level after each call. There is some slight overhead for each kernel call; however, no data is being transferred between the CPU and GPU during this time so the performance impact is minimal.

Following the execution of the algorithm, the original adjacency matrix now contains the distance of the shortest path between each i, j pair and the *next* matrix contains the highest vertex index that i must travel through along the shortest path to j. The shortest path distance information stored in the original adjacency matrix is no longer needed for routing and is therefore discarded. The information that is needed is in the *next* matrix. However it is still not in the correct form. We are looking for next hop information and it is currently giving us a node that is potentially many hops down the line. However, as mentioned in the discussion of the Floyd-Warshall algorithm, next hop information for each pair can be obtained using the *next* matrix. This next hop information for each pair can be determined independently and therefore we use another CUDA kernel program to perform this.

Now that a next hop for each pair of vertices has been calculated, the *next* matrix is copied back from the GPU to the CPU and passed back from the shared library to the helper. The helper then stores it in a location that is globally accessible to all nodes participating in the global routing.

4.2 Actual ns-3 Routing

Like all routing protocols in ns-3, our Ipv4FwCudaRouting module derives from the ns3::Ipv4RoutingProtocol[1] base class and therefore implements RouteOutput and RouteInput. RouteOutput is called when a packet is provided by the transport layer and provides a route towards the destination. RouteInput is called when a packet arrives at a NetDevice and a decision must be made to either forward the packet onward or pass it up to the transport layer.

For RouteOutput and in the case where RouteInput is forwarding a packet, an instance of ns3::Ipv4Route must be assembled to provide the next hop information for the packet. Along with the source and destination for the packet, the information needed for this object is the output device and the next hop IP address. Our implementation obtains this information as follows. First the destination IP address is looked up in the global IP address to node number translation map to get the destination node number. Then the next hop for the destination node from the current node is looked up in the *next* matrix. Now that we have the next hop node number we use it to index the local routing table for this node to get the correct output device and remote IP address. At this point all of the information needed to route the packet has been obtained.

4.3 Example using FW-CUDA Routing

To the user of the new routing library, the interface is very similar to ns-3's current Ipv4GlobalRouting interface. A single call to Ipv4FwCudaHelper::CalculateRoutes() performs all of the route computations.

```
#include "ns3/ipv4-fw-cuda-routing.h"
#include "ns3/ipv4-fw-cuda-helper.h"

Ipv4ListRoutingHelper listRouting;
Ipv4StaticRoutingHelper staticRouting;
Ipv4FwCudaHelper cudaRouting;

listRouting.Add(staticRouting, 0);
listRouting.Add(cudaRouting, 10);

InternetStackHelper stack;
stack.SetRoutingHelper(listRouting);
```

```
//Set up node topology and install stack
...
//Calculate routes
Ipv4FwCudaHelper::CalculateRoutes();

// Run the simulation
Simulator::Run ();
```

5. EXPERIMENTS

All of the experiments performed in this section were run on the Georgia Tech PACE computing cluster. The CPUs on the nodes we used are six-core AMD Opteron Processors running at 2.4 GHz. Each node has a total of 64 GB of available RAM. The GPUs used were T10 Tesla Processors with Compute Capability 1.3 and a total of 4 GB of available global memory.

For the first experiment we wanted to see how the current Ipv4 global routing implementation in ns-3 would perform with a BRITE generated topology with a large number of nodes. For this experiment we used the BRITE hierarchical model with the AS-Barbasi-Albert model as the top level and used the Waxman model for the routers in each AS. We generated a total of 10 AS and the total number of nodes per AS was varied on each run. For each AS, we attached a node containing a UDP packet sink and a UDP source to the first leaf node. Each source was configured to send one packet of 100 bytes to every other sink, therefore each sink should receive 9 packets or 900 bytes. We did not set a stop time for the simulation, therefore it will run until the event list is exhausted. The complexity of the setup is admittedly simple, however the purpose of the experiment is to measure the time needed to generate the routing tables for the topology. The ability to send and receive packets afterwards shows that the process was able to complete and therefore the nodes have the information they need to route packets.

Figure 5 shows the results of this experiment. We varied the number of routers per AS from 100 to 400, thus generating networks varying from 1,000 to 4,000 nodes. We had originally hoped to go even larger; however, we began to run into wall clock time limits on the server on which the program was running. The results clearly show that there is a prohibitive cost when using the current global routing protocol with a large number of nodes. Even with the smallest topology we tested, 1000 nodes, the time to run a simulation which sends 90 packets is almost a 1/2 hour (1566 seconds).

For the second experiment we wanted to compare how our new CUDA global routing module compared to ns-3's Ipv4 global routing and also how it compared to Nix-Vector routing. Nix-Vector[12] routing does an on demand BFS to find the shortest path from source to sink. It then stores the steering information to guide the packet along that path within the packet itself. The node that generates the Nix-Vector also caches it so future transfers are done without the need for another BFS. For this experiment we used the same setup as in the previous experiment. We ran each iteration of the experiment ten times and took an average of the results. The results can be seen in figure 6.

The results show that both Nix-Vector routing and CUDA global routing clearly outperform the current global routing implementation for large topologies in ns-3. For 4000 nodes the run time for global routing was three orders of magnitude higher. Also, the results showed that our CUDA routing protocol outperformed Nix-Vector routing by between 3 and 3.5x for every topology size we tested. The important thing to remember here is what is being calculated in each simulation. In the Nix-Vector experiment, the shortest path is being calculated for ten pairs of nodes; however, in the CUDA routing experiment the shortest path is being calculated for every pair of nodes in the topology.

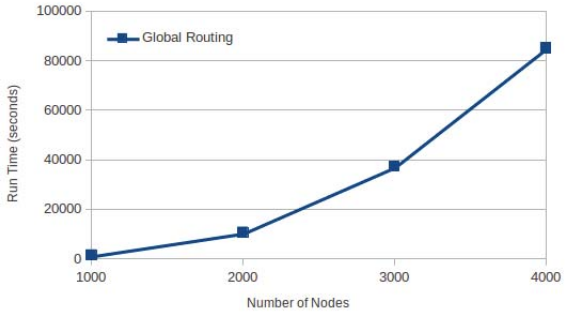


Figure 5: The run time of our test simulation varying the total number of nodes in the topology using Ipv4GlobalRoutingHelper::PopulateRoutingTables().

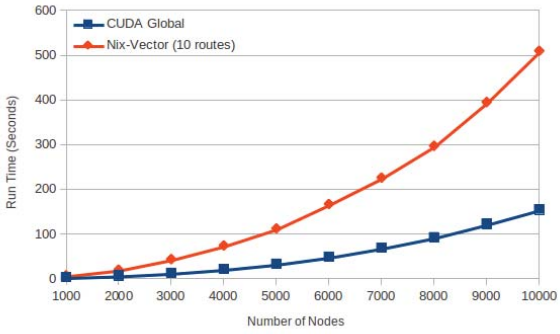


Figure 6: The run time of our experiment varying the total number of nodes in the topology using Nix-Vector routing and our CUDA global routing.

For the third experiment we fixed the number of nodes in the BRITE topology to 5,000. The same hierarchical model that was used in the previous two experiments was used. In addition, we then added a variable number of sending and receiving pairs to leaves in the generated topology. The minimum number of pairs we used was 10 and the maximum was 1000. Each sender was configured to send one packet via TCP to its receiver. By doing this we were able to vary the number of routes that the Nix-Vector protocol needed to calculate. Each configuration was run ten times and the results were averaged. The results can be seen in figure 7. As expected, for both experiments as the number of senders and receivers was increased, the run time of the experiments also increased. This is expected because the simulation has to handle more packet traffic and a greater number of nodes. However, as the results in figure 7 show, this has a much greater effect on the Nix-Vector protocol.

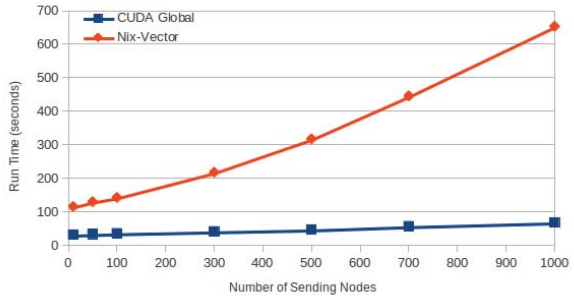


Figure 7: The run time of our experiment varying the number of TCP senders using Nix-Vector routing and our CUDA global routing.

6. NEXT STEPS AND FUTURE WORK

Code optimization on GPU kernel code is a non-trivial process and can have a dramatic effect on overall performance. We have little doubt that our code could be improved by better use of memory coalescing and use of shared memory. These issues will be examined more closely as we continue to test and improve our implementation. Also, before the code could be merged into ns-3, it would need to go through a code review where we are sure we would receive constructive suggestions which would lead to a better overall product and easier integration into the code base.

We are currently working on a SSSP kernel that can be used to generate routing information for a node on demand. Depending on the performance, this could also be used with other ns-3 routing protocols. Also this will allow us to compare a multi-SSSP approach to a APSP approach where both are being used to calculate actual routes.

Furthermore there a few other improvements that can be made to our module. Currently we are including nodes with only a single edge in our adjacency graph. Obviously unless the node is sending to itself, all traffic is going to be going out its only edge. Therefore for each node like this, we could remove a row from both the adjacency matrix and the next matrix. Currently we are mapping a node to a row in each matrix using the node id, node 15 is the 15th row etc, so this would have to be modified for this to work. However, the change would be minor.

7. CONCLUSIONS

In this paper we presented our new BRITE module which allows ns-3 users to take advantage of the topology generation features of the BRITE topology generator. Furthermore, we presented a new CUDA driven global routing protocol which demonstrated substantial speedup compared to the current global routing protocol in ns-3 when simulating large topologies. Our new protocol also showed a significant speedup compared to Nix-Vector routing despite the fact our protocol is generating many more routes.

8. ACKNOWLEDGEMENTS

We wish to thank Tom Henderson for performing the peer-review on our BRITE module. Also we would like to thank Josh Pelkey and John Abraham for their contributions to the BRITE module. Thanks also to Talal Jaafar who wrote the BRITE wrapper for GTNetS on which much of the work on the ns-3 version has been based.

9. REFERENCES

[1] *ns-3 manual*, 2012.

[2] Barabási, A.-L., and Albert, R. Emergence of Scaling in Random Networks. *Science 286*, 5439 (Oct. 1999), 509–512.

[3] Buluç, A., Gilbert, J. R., and Budak, C. Solving path problems on the gpu. *Parallel Comput. 36*, 5-6 (June 2010), 241–253.

[4] Calvert, K., Doar, M. B., Nexion, A., Zegura, E. W., Tech, G., and Tech, G. Modeling internet topology. *IEEE Communications Magazine 35* (1997), 160–163.

[5] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[6] ErdÅŚs, P., and RÃL'nyi, A. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES* (1960), pp. 17–61.

[7] Harish, P., and Narayanan, P. J. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing* (Berlin, Heidelberg, 2007), HiPC'07, Springer-Verlag, pp. 197–208.

[8] Katz, G. J., and Kider, Jr, J. T. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2008), GH '08, Eurographics Association, pp. 47–55.

[9] Lund, B. D., and Smith, J. W. A multi-stage cuda kernel for floyd-warshall. *CoRR abs/1001.4108* (2010).

[10] Medina, A., Matta, I., and Byers, J. Brite: A flexible generator of internet topologies. Tech. rep., Boston, MA, USA, 2000.

[11] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[12] Riley, G. F., Ammar, M. H., and Fujimoto, R. M. Stateless routing in network simulations. In *Proceedings of the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (August 2000).

[13] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 73–82.

[14] Venkataraman, G., Sahni, S., and Mukhopadhyaya, S. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics 8* (Dec. 2003).

[15] Waxman, B. Routing of multipoint connections. *Selected Areas in Communications, IEEE Journal on 6*, 9 (dec 1988), 1617 –1622.