GPU-Assisted Hybrid Network Traffic Model

Jason Liu
Florida International University
Miami, Florida, USA
liux@cis.fiu.edu

Yuan Liu
Tsinghua University
Beijing, China
liuy139@gmail.com

Zhihui Du
Tsinghua University
Beijing, China
duzh@tsinghua.edu.cn

Ting Li
Florida International University
Miami, Florida, USA
tli001@cis.fiu.edu

ABSTRACT

Large-scale network simulation imposes extremely high computing demand. While parallel processing techniques allows network simulation to scale up and benefit from contemporary high-end computing platforms, multi-resolutional modeling techniques, which differentiate network traffic representations in network models, can substantially reduce the computational requirement. In this paper, we present a novel method for offloading computationally intensive bulk traffic calculations to the background onto GPU, while leaving CPU to simulate detailed network transactions in the foreground. We present a hybrid traffic model that combines the foreground packet-oriented discrete-event simulation on CPU with the background fluid-based numerical calculations on GPU. In particular, we present several optimizations to efficiently integrate packet and fluid flows in simulation with overlapping computations on CPU and GPU. These optimizations exploit the lookahead inherent to the fluid equations, and take advantage of batch runs with fix-up computation and on-demand prefetching to reduce the frequency of interactions between CPU and GPU. Experiments show that our GPU-assisted hybrid traffic model can achieve substantial performance improvement over the CPU-only approach, while still maintaining good accuracy.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques; I.6.3 [**Simulation and Modeling**]: Applications

Keywords

GPU, network simulation, fluid model

1. INTRODUCTION

Simulation is an effective method for studying network protocols and applications, capable of capturing detailed operations of a complex network, especially the cross-layer interoperation of various network protocols and components that are difficult to reproduce in real network testbeds. Simulation, once validated, can be especially cost effective for studying new network designs and services by offering controlled, diverse, and yet reasonably realistic network scenarios before the actual deployment.

The computational requirement of simulating large-scale network can be extremely high. Consider an example of simulating the core network of a major US Internet service provider, in this case, AS 7018 from the RocketFuel dataset, which consists of about 12,000 routers and 15,000 links [12]. Assuming all links are gigabit connections and assuming the average packet size is 500 bytes, in order to simulate this network with a mere 10% utilization, we can have a back-of-an-envelop calculation which shows that the simulator would need to process on average about 375 million packet events per second. (A packet event is defined as a simulation event representing a packet either arriving at or departing from a host or a router, and can be considered as the unit of simulation workload.) To meet with such computational demand, there are two general approaches.

In one approach we can use parallel simulation to harness the collective computing power of parallel machines [4]. In this case a large network is partitioned among the available processors and cores; each submodel is a logical process that maintains its own event list and simulation clock, and can run on a separate processor. Each logical process must process local simulation events independently in timestamp order, and synchronize and communicate with other logical processes via timestamped messages. Several parallel network simulators (e.g., [3, 32, 40]) have demonstrated the potential of running large network models on massively parallel computers.

In another approach we can increase the level of abstraction in order to reduce the computational demand of a large-scale network simulation. Fluid traffic models provide a first-order approximation of the aggregate network behavior by capturing the flow rates rather than individual packets. A discrete-event formulation of a fluid flow can describe the changes in the flow rate as the flow traverses the network links and competes for network resources with other traffic [19]. Alternatively, one can use differential equations to represent the behavior of persistent TCP flows and their effect on the network queue length [16]. These differential equations can be numerically solved with efficiency. In both

SIGSIM-PADS'14, May 18–21, 2014, Denver, CO, USA.

http://dx.doi.org/10.1145/2601381.2601382.

cases the fluid models can demonstratively achieve a speedup over packet-oriented simulation by as much as three orders of magnitude while maintaining good accuracy.

This gives rise to an important distinction between the foreground and background traffic. The foreground traffic consists of detailed transactions of the network protocols and applications that we intend to study and therefore need to be modeled with high fidelity. The background traffic is the stochastic processes that govern the behavior of the bulk of the network traffic that occupies the network links. The background traffic causes the characteristic fluctuations of the network queues and therefore affects the packet delays and losses of the foreground traffic. Since the background traffic does not require significant accuracy, we can model the background traffic as fluid flows. Previously we developed a hybrid network traffic model that integrates the packet-oriented foreground traffic, which is simulated using discrete-event simulation, and the fluid-based background traffic, which is described by a set of differential equations and solved numerically using a time-stepped approach [13].

Once a specialized processing unit dedicated only for graphics rendering, the Graphic Processing Unit (GPU) has now become an important massively parallel computing platform, suitable for high-performance high-throughput data parallel computations. Currently the performance gap between GPU and CPU is roughly 10x, both in the processing rate and the main memory bandwidth. Such a gap is expected to widen in the next decade. Nowadays, GPU is commonly available among workstations; because of its low cost performance ratio and energy efficiency, GPU is already making regular appearances in high-profile enterprise systems and supercomputers. For example, according to the November 2013 list of the top 500 supercomputers [36], although less than 10% of the current supercomputer systems come with GPU, statistics show that nearly one third of the overall performance of these supercomputers has been brought by GPU acceleration.

In this paper, we present an extension to the hybrid network traffic model, which offloads the numerically intensive background traffic calculations to GPU. We propose an integration scheme that can overlap the CPU-based discrete-event simulation of the foreground network packets and the GPU-based differential equation solver for the background fluid dynamics. We describe a novel mechanism that can minimize the effect of inherent communication latencies between CPU and GPU. Our work takes advantage of the GPU capabilities by moving the computationally intensive background traffic calculations to GPU, so that CPU can concentrate on simulating the detailed transactions of network protocols and applications.

The contributions of this paper can be summarized in two folds: (1) We present a hybrid traffic model that cleanly separates packet-oriented foreground traffic simulation that executes on CPU and the fluid-based background traffic calculation that executes on GPU; (2) We propose a set of optimization algorithms for overlapping asynchronous CPU and GPU computations, exploiting the lookahead information in the fluid calculations, and using batch runs and associated fix-up computations to reduce frequent interactions between CPU and GPU. Extensive experiments for validating the GPU-assisted hybrid traffic model using controlled network scenarios, and demonstrate significant performance improvements (as much as 25x) using large network scenarios. In this aspect, our work sets the stage for massive-scale network simulation with realistic traffic behavior on hybrid supercomputing platforms with GPU acceleration.

The rest of the paper is organized as follows. Section 2 reviews the existing work in network traffic modeling and GPU-based simulation and modeling techniques. Section 3 describes the hybrid traffic model and presents the basic method for integrating the packet-oriented simulation on CPU and the fluid-based traffic calculation on GPU. Section 4 describes the optimization techniques that can efficiently overlap CPU and GPU computations and reduce communication overhead between CPU and GPU. Section 5 presents our validation and performance studies. Section 6 concludes the paper and outlines future work.

2. BACKGROUND

In this section we present an overview of fluid network traffic models. We focus on the integration schemes from which we develop our GPU-based solution. We also describe existing work on GPU-based simulation and modeling.

2.1 Fluid Network Traffic Models

Traditional infrastructure network simulations represent network transactions at the packet level. It is computationally expensive since we use at least one simulation event for each packet entering or leaving a network host or router. Modeling network traffic as fluid flows can be traced back to the idea of using packet trains by Ahn and Danzig [2]. Fluid models need to accurately and efficiently capture flow-level characteristics, such as the flow rate, as an approximation of the network effect. For example, Nicol [18] used piecewise linear functions to represent TCP traffic flows and uses simulation events to represent the flow rate changes as the flows are propagated downstream. Guo et al. [7] proposed a time-stepped approach where consecutive packets arriving at a router within a time-step are lumped together and represented by a single flow rate. Recently Li et al. [10] proposed a fast rate-based TCP (RTCP) traffic model that approximates traffic flows as a series of *rate windows*, each consisting of a number of packets considered to possess the same arrival rate.

Misra et al. [17] used a set of ordinary differential equations (ODE) to represent the flow rate changes by capturing the long-term average behavior of persistent TCP flows. Liu et al. [16] later provided several improvements to the Misra's algorithm by explicitly expressing the network topological information using a set of nonlinear time-varying ordinary differential equations, which keep track of the time-varying congestion window size of the fluid flows and the length of the network queues.

These differential equations can be solved numerically using a fixed time-stepped Runge-Kutta method. This can be achieved by having the simulator to schedule an event periodically with a small time interval. At each step, the differential equations are evaluated, and subsequently the congestion window sizes and the queue lengths are updated. To achieve better efficiency, multiple fluid flows with the same source and destination can be consolidated into one flow following the same congestion window trajectory.

2.2 Integration of Packet and Fluid Models

Hybrid network traffic models aim at combining the packet-oriented foreground flows and the fluid-based background

flows within the same simulator, and as such they need to focus on the interaction between the two types of network flows. Discrete-event fluid models (e.g., [18]) use simulation events to capture the moments of flow rate changes and thus can be naturally integrated with the packet-level simulation (e.g., [9, 20, 33]).

Special arrangement, however, must be made to integrate the packet-oriented discrete-event simulation and the fluid model based on differential equations. Gu et al. [6] presented a method of integrating two separate networks: a fluid network at the core with states represented by differential equations, and a packet network at the peripheral with network transactions represented by discrete events. Interactions are allowed to happen only at the network boundary: packets entering the core network must be converted into fluid flows, and when they leave, they must be converted back to packet events. Zhou et al. [41] suggested an improvement. In order to achieve a better response time from the ODE solver (implemented in MATLAB), two instances of the same fluid model are included in their system with interleaving simulation clocks. Such redundant computation can effectively double the speed of the ODE solver.

In our previous work we proposed an integration scheme that allows mixing of fluid and packet flows at each network queue [13]. To correctly integrate the fluid and packet models, the scheme takes into consideration the impact of fluid flows on packet flows and vice versa. In particular, one must calculate the aggregate arrival rate of both fluid and packet flows at each network queue for correct queue length and queuing delay for all fluid flows traversing a node. In doing so, one can properly schedule the packet departure events according to the calculated queuing occupancy.

We also proposed several techniques to improve the performance of the hybrid model, such as using efficient data structures, caching, and dynamically varying the Runge-Kutta time step size [15]. We also demonstrated that the hybrid model is highly parallelizable [14]. We observed that the time it takes to propagate fluid characteristics (such as the accumulative delay and loss rate) along the flow paths has a lower bound equal to the minimum link delay, according to the governing ordinary differential equations. As such, parallel simulation can maintain good lookahead and is thus capable of achieving good parallel performance.

2.3 GPU-based Simulation

Modern GPUs have recently evolved beyond simply being the graphics processing units for rasterization of 3D primitives. It is common that contemporary high-end computing systems include GPUs to accelerate computation. Over the years, we have seen many algorithms and applications taking advantage of GPU's unique processing capabilities [5, 23, 29]. The GPU programming tools have also evolved from specific graphics languages, such as OpenGL and Cg, to more generic parallel programming paradigms, such as CUDA [21] and OpenCL [34].

GPU-based simulation has also become more common. Verdesca et al. [37] used GPU to conduct line-of-sight and route planning calculations for battlefield simulations. It is not surprising that simulation can benefit significantly from offloading data parallel tasks onto GPU, including for example, N-body simulation [8, 22], Monte Carlo simulation [30, 38], group mobility models [28], and agent-based models [1, 27].

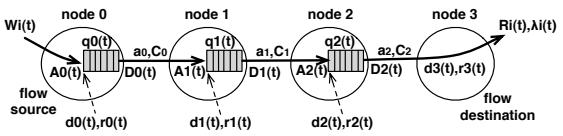

Figure 1: A flow traversing two intermediate nodes.

Perumalla [26] first investigated possible alternatives for conducting discrete event simulation (DES) on GPU. Park and Fishwick [24] developed an application framework based on CUDA to support fast DES. In a consequent paper [25], they conducted an analysis of queuing network simulation using their GPU-based application framework. More recently, Li et al [11] introduced the a three-stage strategy to realize DES on the GPU platform as a cost-efficient alternative to the traditional parallel DES. Tang and Yao [35] developed a new simulation kernel based on GPU to support DES. All the above efforts aimed at accelerating generic discrete-event simulation on GPU. Our method described in this paper is specific for cooperating CPU and GPU for high-performance network traffic modeling.

In this aspect, Xu and Bagrodia [39] presented a hybrid network simulation framework that uses GPU to carry out certain tasks, including a GPU implementation of a fluid TCP model [17]. Our hybrid traffic model is derived from an improved fluid model with detailed network topological information. To achieve better accuracy, our model also requires fine-tuned integration between the fluid and the packet flows, which presents unique challenges for coordinating the computation carried out both on CPU and GPU.

3. HYBRID TRAFFIC MODELING

Our hybrid traffic model consists of a fluid-based background traffic model on GPU, and a discrete-event packet-oriented foreground traffic model on CPU. We describe the fluid model and its GPU-based implementation, and then introduce the integration scheme that combines the GPU-based fluid model and the CPU-based packet simulation.

3.1 Fluid Model

The input to the background fluid traffic model is a graph that represents the network with link delays and bandwidths, and a traffic matrix consisting of fluid flows on the network with distinct sources and destinations as the background traffic. In the following we first describe the set of differential equations that need to be solved on GPU. Compared with the original fluid model [16], here we also present a simplified model for drop-tail queues in addition to the Random Early Detection (RED) queuing policy. Fig. 1 illustrates an example with a fluid flow traversing two intermediate nodes; it shows most of the flow variables maintained by the model.

For TCP flow i, the additive-increase and multiplicative-decrease behavior of the TCP window size $W_i(t)$ (during the TCP congestion avoidance stage) can be described as:

$$dW_i(t)/dt = 1/R_i(t) - W_i(t)\lambda_i(t)/2 \quad (1)$$

where $R_i(t)$ is the round-trip delay and $\lambda_i(t)$ is the packet loss rate at time t.[1]

[1]The boundary conditions of the window size and the queue length (shown later) are not included here in the equations for simple exposition.

Suppose flow i visits a total of n_i nodes from source to destination, where node 0 is the source and node $(n_i - 1)$ is the destination of the flow. At each node k except the destination node, we can calculate its queue length $q_k(t)$ using the following differential equation:

$$dq_k(t)/dt = \Lambda_k(t)(1 - \rho_k(t)) - C_k \quad (2)$$

where$\Lambda_{\ k}(t)$ is the total arrival rate of all fluid flows entering the network queue, $\rho_k(t)$ is the packet drop probability, and C_k is the link bandwidth. The packet drop probability $\rho_k(t)$ is used specifically to model the selective packet dropping mechanism in RED. For drop-tail queues, we set $\rho_k(t)$ to zero. For RED, it is a piece-wise linear function of the average queue length, which can be calculated as a moving average from the instant queue length.

The round trip time, $R_i(t)$, and the packet loss rate, $\lambda_i(t)$, are determined by accumulating queuing delays and packet losses along the flow path. We use $d_k(t)$ to denote the cumulative delay, and $r_k(t)$ to denote the cumulative packet loss rate of flow i arriving at the k^{th} node, where $0 \leq k < n_i$. At the source, both $d_0(t)$ and $r_0(t)$ are zero.

The cumulative delay at the downstream node $(k+1)$ can be calculated from the value at node k:

$$d_{k+1}(t_f) = d_k(t) + a_k + q_k(t)/C_k \quad (3)$$

where $t_f = t + a_k + q_k(t)/C_k$, and a_k is the link propagation delay between node k and node $k+1$. Here, the cumulative delay at the downstream node is calculated from the cumulative delay at the previous node plus the queuing delay at the previous node and the link's propagation delay. Note that the cumulative delay must consider a time lag equal to the sum of the queuing delay and the propagation delay.

The cumulative packet loss rate for the fluid flow arriving at the downstream node $(k+1)$ can be calculated from the cumulative packet loss rate at node k plus all the losses occurred at node k (also with the proper time lag). For RED queues, this can be expressed as:

$$r_{k+1}(t_f) = r_k(t) + A_k(t)\rho_k(t) \quad (4)$$

where $A_k(t)$ is the arrival rate and $\rho_k(t)$ is the RED packet drop probability at node k. For drop-tail queues, we have:

$$r_{k+1}(t_f) = r_k(t) + A_k(t) - D_k(t + q_k(t)/C_k) \quad (5)$$

Here, $A_k(t)$ is the arrival rate of the fluid flow and $D_k(t + q_k(t)/C_k)$ is departure rate of the same flow. The difference accounts for the loss.

At the flow source, the arrival rate is the flow send rate, which can be calculated from the TCP congestion window size and the round-trip delay:

$$A_0(t) = W_i(t)/R_i(t) \quad (6)$$

Similar to [16], we allow a fluid flow to include multiple TCP sessions having the same source and destination and therefore following the same congestion window trajectory. For that, we simply multiply the send rate by the number of TCP sessions. Note that if flow i is an UDP flow, we simply set a constant send rate.

For subsequent nodes, the arrival rate is the departure rate of the previous queue after a time lag equal to the link's propagation delay:

$$A_{k+1}(t + a_k) = D_k(t) \quad (7)$$

The departure rate at node k is determined by the arrival rate after a time lag equal to the queuing delay, which amounts to $q_k(t)/C_k$. If the total arrival rate is less than the bandwidth, the departure rate remains the same as the arrival rate. Otherwise, the departure rate is proportional to the arrival rate as the bandwidth is shared among the competing flows:

$$D_k(t'_f) = \begin{cases} A_k(t)(1 - \rho_k(t)) & \text{if}\Lambda_{\ k}(t)(1 - \rho_k(t)) \leq C_k \\ A_k(t)C_k/\Lambda_k(t) & \text{otherwise.} \end{cases} \quad (8)$$

where $t'_f = t + q_k(t)/C_k$.

For simplicity, we assume the routing path is symmetrical, and the queuing delays and packet losses are negligible for the ACK flow from the destination traveling back to the source. Let π_i be path delay for flow i, which is the sum of the propagation delay of all links on the path from source to destination. The round-trip delay can be calculated from the cumulative delay at the flow destination:

$$R_i(t) = d_{n_i-1}(t - \pi_i) + \pi_i \quad (9)$$

Similarly, the packet loss rate can also be calculated from the cumulative packet loss rate at the flow destination:

$$\lambda_i(t) = r_{n_i-1}(t - \pi_i) \quad (10)$$

In case a flow consists of multiple TCP sessions, we need to divide the total loss rate by the number of TCP sessions to derive the *per-session* loss rate, which we use in Eqn. (1).

The above equations can be solved numerically using the Runge-Kutta method. In general, to solve $y(t)$ given $dy(t)/dt$, and the initial value $y(0) = y_0$, one can iteratively compute the approximate values $y_1, y_2, \cdots, y_n$ of the actual values $y(t_1), y(t_2), \cdots, y(t_n)$, where $t_0 = 0$ and $t_{i+1} = t_i + \delta$ for $i = 0, 1, \cdots, n-1$. δ is the Runge-Kutta step size, which is set to be at least 10x smaller than the link delay in the implementation in order to maintain numerical accuracy.

In parallel simulation, a fluid flow may traverse multiple sub-networks each assigned to a different processor. We use ghost nodes to represent the next fluid nodes assigned to remote processors. The ghost nodes communicate with the remote processors by the underlying parallel simulation kernel. The fluid nodes along the path of a fluid flow are thus broken into multiple segments assigned to different processors and handled in a parallel fashion. There is a full discussion in [14].

3.2 GPU Implementation

We implement the Runge-Kutta method on GPU using CUDA [21]. In CUDA, a program is composed of a large number concurrently schedulable threads. These threads are organized into thread blocks and dispatched onto GPU's parallel execution units, called Streaming Multiprocessors (SMs), via invocation of the CUDA "kernel" functions. While threads within the same block can synchronize using CUDA's `__syncthreads()` function, synchronization between threads belonging to different blocks is unsupported. In this case, one would have to split calculations into serval kernels so that data modification by threads in one kernel are visible to threads in subsequent kernels.

There exist data dependencies during the evaluation of the differential equations within a Runge-Kutta step. For example, the departure rate at the source of the fluid flow depends on the flow send rate (as well as the queue length),

which in turn depends on the congestion window size and the round-trip time. We split the fluid calculations into three kernels. We make sure there is no data dependency within each kernel; therefore, we can employ more threads and assign them to multiple thread blocks for better parallelism.

The first kernel function, `update_flow`, designates a CUDA thread for evaluating the congestion window size $W_i(\cdot)$, the round-trip time $R_i(\cdot)$, and the packet loss rate $\lambda_i(\cdot)$, for each flow i. The second kernel function, `update_queue`, designates a CUDA thread for calculating the queue length $q_l(\cdot)$ for each network queue l that contains fluid flows. Here, each thread performs a gather operation to compute the total arrival rate of all fluid flows entering the queue. The third kernel function, `update_hop`, designates a CUDA thread for each fluid flow at a network queue that the fluid flow traverses. Each thread needs to evaluate the flow arrival rate $A_k(\cdot)$, the flow departure rate $D_k(\cdot)$, the cumulative delay $d_k(\cdot)$, and the cumulative packet loss rate $r_k(\cdot)$. At each Runge-Kutta step, these three kernels will be invoked in sequence. Data transfer between CPU and GPU will happen before and after the kernel invocations. It is obviously inefficient to invoke the kernels and transfer data at such frequency; we discuss further optimizations in Section 4.

CUDA provides users access to different types of memory. Each thread may use *registers* or *shared memory* for fastest access. The latter can be used to communicate with threads within the same block. However, both types of memory are not persistent across kernel invocations. There are also *constant memory* and *texture memory*, but they are used for special purposes: constant memory is used only for storing immutable data, and texture memory requires 2D spatial locality.

Since fluid variables, like the congestion window size $W_i(\cdot)$ and the cumulative packet loss rate $r_k(\cdot)$, need to be kept at GPU across kernel invocations, and they also need to be accessed by threads potentially belonging to different blocks, we use *global memory*, which is a large chunk of memory and can be accessed by all threads belonging to different blocks and across kernel invocations. Although accessing the global memory is about 100x slower than the registers and shared memory, it is still considered to have a higher bandwidth than the CPU memory.

Certain fluid variables need to keep values in the simulated future. For example, the cumulative delay $d_k(\cdot)$ in Eqn. (3) must be able to set values at a future time after the current queuing delay and the link propagation delay. In this case, we allocate memory for an array to keep track of the time series. The size of the array is bounded by the maximum queuing delay and the link propagation delay.

3.3 Packet and Fluid Integration

The fluid background traffic calculation on GPU is invoked as CUDA kernel functions by CPU at each Runge-Kutta step. However, the GPU-based fluid model only considers fluid flows; these fluid flows need to be integrated with the foreground traffic, which is represented by individual packets and simulated as discrete events on CPU.

In particular, when a packet arrives at a node, the simulator needs to determine whether this packet will cause the network queue to overflow with the given queue length calculated from the background fluid model. If so, the packet needs to be dropped. Otherwise, it is inserted into the network queue and the network queue length is adjusted accordingly. The simulator also needs to schedule a packet departure event after the queuing delay. The fluid model needs to consider the changes to the network queue length due to packet arrivals. The fluid model only calculates the differences in queue length at each Runge-Kutta step (using Eqn. 2).

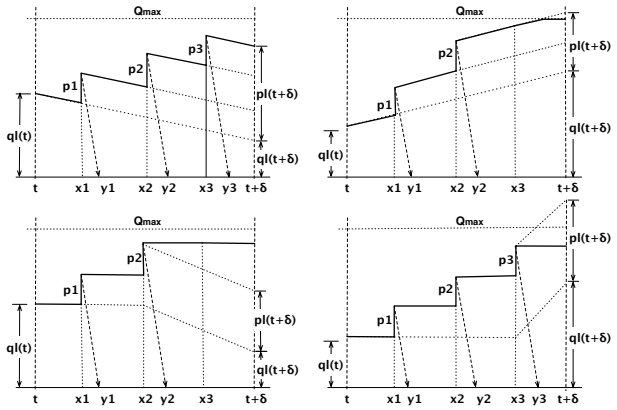

Figure 2: Mixing of fluids and packets.

Fig. 2 shows an example on how fluids and packets are integrated. Suppose at time t and $t+\delta$ (δ is the Runge-Kutta step size), the GPU-based fluid model calculates the network queue length $q_l(t)$ and $q_l(t+\delta)$. The top left plot shows the situation where the aggregate fluid arrival rate is less than the bandwidth and therefore the queue length is decreasing. The top right plot shows the fluid arrival is greater than the bandwidth, and the queue length is increasing. During the time interval, three packets of size p_1, p_2, and p_3 arrive at the network queue at time x_1, x_2, and x_3, respectively. The top left plot shows that all packets are admitted into the network queue; however, the top right plot shows that the third packet is dropped due to queue overflow.

When a packet is inserted into the queue, we need to adjust the queue length. We use variable $p_l(x)$ to keep track of the total size of packets arrived at the queue during the interval before and including time x. We set $p_l(t) = 0$ at the start of the interval. If a packet is allowed to be inserted into the queue upon its arrival at time x, the packet size will be added to $p_l(x)$ and then the simulator will schedule a packet departure event so that it can later simulate the delivery of the packet to the next queue. The departure time y can be calculated from the arrival time x and the current queuing delay, which depends on the length of the queue (including for both fluids and packets) and the link bandwidth C_l:

$$y = x + (q_l(x) + p_l(x))/C_l \tag{11}$$

The problem is that when packets arrive during the Runge-Kutta time interval between time t and $t+\delta$, the fluid queue length at time $t+\delta$ is not yet available (it's in the simulated future). As a matter of fact, $q_l(t+\delta)$ actually should depend on both fluid and packet arrivals during the interval. In our implementation, we approximate the queuing occupancy by assuming the fluid queue length $q_l(\cdot)$ stays constant during the interval.

This approximation may introduce errors, which are illustrated in the bottom two plots in Fig. 2. The bottom left plot shows the situation where the third packet, which is supposed to be admitted, is dropped due to queue overflow. And the bottom right plot shows the third packet is admitted,

Algorithm 1 Lock-step hybrid simulation on CPU

```
1: q_l, p_l ⇐ 0 for all network queues
2: schedule a Runge-Kutta event at time 0
3: ··· // initialize other simulation variables
4: While (simulation is not finished) Do
5:    e ⇐ eventlist.getEarliestEvent()
6:    t_c ⇐ e.time // current simulation time
7:    If (e is a Runge-Kutta event) Then
8:       q_l ⇐ q_l + p_l; p_l ⇐ 0 for all network queues
9:       copy q_l for all network queues CPU⇒GPU
10:      invoke update_flow, update_queue, update_hop
11:      wait for the three kernels to complete
12:      copy q_l and ρ_l for all network queues GPU⇒CPU
13:      schedule a Runge-Kutta event at time t_c + δ
14:   Else If (e is a packet arrival event) Then
15:      u ⇐ uniform(0,1)
16:      If (u < ρ_l OR q_l + p_l + packet.size > Q_l^max) Then
17:         drop the packet
18:      Else
19:         insert packet into the queue
20:         p_l ⇐ p_l+ packet.size
21:         schedule packet departure event at time t_c+(q_l+
            p_l)/C_l
22:      End If
23:   Else
24:      ··· // process other simulation events
25:   End If
26: End While
```

but in reality it should be dropped. Similarly, we assume the packet drop probability for the REQ queue $\rho_l(\cdot)$ also stays constant during the interval. The packet drop probability is calculated from the average queue length, which could change during the interval when the instant queue length changes. In both cases, however, we expect the error to be insignificant. The Runge-Kutta step size must be kept small to maintain numerical accuracy. The differences only occur when the network queue is about to be full. When we keep the Runge-Kutta step size small, the chance of having multiple packets to arrive during the interval and causing such error cannot be significant.

The algorithm for integrating fluid flows and packet flows is described in detail in Alg. 1. At each Runge-Kutta step, CPU needs to copy the updated queue length of all network queues to GPU global memory (at lines 8 and 9). The CUDA kernel functions are then invoked, which calculate the flow values of the current time, which include the congestion window size, the queue length, and the cumulative delay and packet loss rate (at lines 10 and 11). CPU then transfers the queue length and packet drop probability from GPU's global memory back to CPU's main memory (at line 12), so that it is ready for the next Runge-Kutta interval.

Upon each packet arrival, the simulator first draws a random number from the uniform distribution between 0 and 1 (at line 15). If it's less than ρ_l, which means the packet is randomly picked to be dropped according to the RED queuing policy, or if the packet would cause to queue to overflow (at line 16), the simulator drops the packet. Otherwise, the simulator inserts the packet into the queue (at line 19), updates the queue size (at line 20), and then schedules the packet departure with the proper delay (at line 21).

Note that, in this algorithm, the invocation of the GPU kernel functions is synchronous. CPU needs to wait for the GPU kernel functions to complete (at line 11) before it can continue with simulation for the next Runge-Kutta interval. This algorithm therefore is *lock-stepped*: there is no overlap between CPU and GPU computations. In the next section, we describe several optimizations for CPU and GPU to perform their tasks asynchronously and more efficiently.

4. OPTIMIZATIONS

In the previous section, we introduce a hybrid traffic model which performs the background fluid-based traffic calculation on GPU and the foreground packet-oriented simulation on CPU. The algorithm is inefficient, however, due to the strong coupling between CPU and GPU computations—at each Runge-Kutta step, CPU has to transfer data to GPU, wait for the kernel invocations to complete, and then transfer data back from GPU, before it can continue with the discrete-event simulation for the next interval. In this section, we present optimizations to allow overlapping CPU and GPU calculations with reduced synchronization frequency.

4.1 Exploiting Lookahead

A close inspection on the set of differential equations of the fluid model reveals that the GPU threads evaluating the equations can be made independent of one another for a time interval larger than the Runge-Kutta step size. We observe that the flow values at one queue does not influence the calculation of the flow values of another queue for a period of time no less than the propagation delay of the link in-between.

Specifically, we see that , according to Eqn. (7), the arrival rate at the downstream queue $A_{k+1}(\cdot)$ is only dependent on the departure rate of the previous queue $D_k(\cdot)$ after a time lag equal to the link's propagation delay a_k. Also, in Eqns. (3), (4), and (5), we see that the the cumulative delay $d_{k+1}(\cdot)$ and the cumulative packet loss rate $r_{k+1}(\cdot)$ at the downstream queue are all dependent on corresponding values at the predecessor queue after a time lag equal to the sum of the link's propagation delay and the queuing delay, $a_k + q_k(t)/C_k$. Similarly, in Eqns. (9) and (10), the round-trip delay $R_i(\cdot)$ and the packet loss rate $\lambda_i(\cdot)$ are calculated from the cumulative delay and the cumulative packet loss rate, respectively, with a time lag as large as the path delay π_i.

This means that the thread responsible for evaluating the flow variables associated with a queue can run independently from the other threads within the same time window of size equal to the minimum propagation delay. Note that this concept is similar to the lookahead we use for parallel simulation. Lookahead is defined to be the minimum simulation time it takes for one simulation process to affect the state of another. With good lookahead, a process can safely process simulation events and advance its simulation clock in parallel with the other processes.

In this context, a simulation process is analogous to a GPU thread. To exploit this lookahead, we still keep the three GPU kernel functions, `update_flow`, `update_queue`, and `update_hop`; however, we make the threads to evaluate for as many as τ number of Runge-Kutta steps at each kernel invocation, where $\tau = \min\{a_k\}/\delta$, and δ is the Runge-Kutta step size. Consequently, data transfer between CPU and GPU happens only at each kernel invocation every $\tau\delta$ units of time. Given that the minimum link propagation delay is at least 10 times larger than the Runge-Kutta step size (for numerical stability), this optimization can significantly reduce the overhead associated with the kernel invocations and data transfers between CPU and GPU.

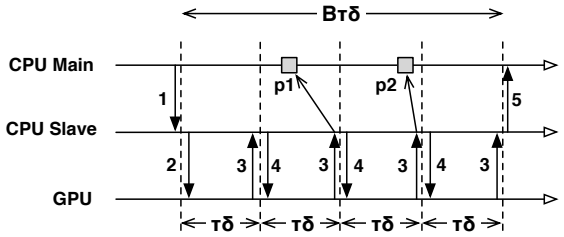

Figure 3: Overlapping CPU/GPU computation.

4.2 Overlapping CPU and GPU Computations

GPU is a separate computing device and can be treated as a co-processing unit to CPU. To improve performance, we need to overlap CPU and GPU computations and have CPU and GPU to perform tasks in parallel. For our hybrid traffic model, this means we want CPU to process simulation events associated with the foreground traffic and GPU to calculate the background traffic, *concurrently*.

We solve this problem by creating a separate thread on CPU to handle GPU kernel invocations for GPU to coast forward for a batch of Runge-Kutta steps without further update from CPU. As illustrated in Fig. 3, the method maintains two threads on CPU: the main thread for processing simulation events, and the slave thread for invoking the GPU kernel functions. The following steps corresponds to the numbers shown in the figure:

1. The CPU main thread schedules a Runge-Kutta batch event periodically every $B\tau\delta$ units of simulation, where B is the batch size, τ is determined by the lookahead described in the previous section, and δ is the Rung-Kutta step size. When processing this event, the main thread first check that the slave thread has completed the previous batch run and if so, signals the slave thread to set off a new batch run. CPU will continue processing events until the next Runge-Kutta batch event.
2. The CPU slave thread waits for the signal from the main thread to start a batch run. Upon receiving this signal, the slave thread transfers the updated queue lengths for all network queues from CPU to GPU. It then invokes the three GPU kernel functions in order: `update_flow`, `update_queue`, and `update_hop`. Each thread will run for τ number of Runge-Kutta steps.
3. The CPU slave thread waits for all three GPU kernels to complete and then copies the resulting fluid queue values from GPU. The CPU main thread processes packet arrival events (p_1 and p_2 in the figure) using the fluid queue values of the corresponding Runge-Kutta step, like in Alg. 1. This means that GPU may need to run ahead of CPU by as much as $\tau\delta$ time. In other words, the CPU main thread may need to wait for the slave thread (and the GPU) to get ahead. Synchronization between the CPU main thread and slave thread can be easily implemented using thread conditional variables.
4. The CPU slave thread sets off for the next τ number of Runge-Kutta steps by invoking the three GPU kernel functions. This step is similar to step 2, except here it does not include data transfers from CPU to GPU. Steps 3 and 4 alternates until GPU completes the whole batch.
5. The CPU slave thread signals the main thread that the batch run has completed, so that the main thread can start with the next batch run.

The above method allows GPU to run the fluid model for a batch of $B\tau$ steps using the updated queue lengths received from CPU at the beginning each batch (in step 2). The resulting fluid values (fluid queue length and drop probability) are copied back to CPU every τ steps. In the next section, we introduce a fix-up computation to avoid the accumulation of errors. In another section to follow, we also introduce an on-demand prefetching technique to further reduce data transfers during a batch run.

4.3 Fix-up Computation

In Section 3.3, we introduce an approximation method for mixing fluid and packet flows within a Runge-Kutta interval. We observe that the approximation may introduce errors due to possible queue overflow. Here we call it an *overflow problem*. We expect the overflow problem should not be significant because of the small step size.

When we consider multiple Runge-Kutta steps for batch runs (as detailed in the previous section), we face two other problems. One problem is how to set the batch size B. The fluid model on GPU only receives an update of the queue length from CPU at the beginning of each batch run. It then carries out the calculations for the next $B\tau$ number of Runge-Kutta steps independent of the changes happen at CPU. This is the same as to assume that the fluid traffic may not be significantly influenced by the packet arrivals during this time period. In general, the background traffic is the dominant traffic on the network, in which case the influence of packet flows on fluid flows is not as important as the other way around. Setting a bigger batch size can make fluid traffic to be less responsive to the changes in the packet flows. One must determine whether this situation is desirable for the simulation problem at hand, and set the batch size cognizant of the performance and accuracy tradeoff.

The other problem we face when dealing with batch runs is what we call an *underflow problem*, an example of which is illustrated by the top plot of Fig. 4. The figure shows the network queue size fluctuates as a function of fluid and packet arrivals over time. For a batch run that starts at time t_B, GPU calculates the background traffic flows and their effect on the fluid queue length, $q_l(\cdot)$, at time $t_B + \delta, t_B + 2\delta, \cdots, t_B + B\tau\delta$. For packet arrivals, the simulator accumulates the packet queue length $p_l(\cdot)$, as shown in Alg. 1 (at line 20). The sum of fluid and packet queue lengths determines the departure time of the arrived packet (see Eqn. 11). The underflow problem is that this method can gravely over-estimate the actual queue length when dealing batch runs having multiple Runge-Kutta steps.

The error occurs at the time when the fluid queue length gets close to zero. The fluid model calculates the queue length, $q_l(\cdot)$, according to the aggregate arrival rate and the bandwidth (Eqn. 2). When the queue becomes empty, for example, during the interval between $t_B + 2\delta$ and $t_B + 3\delta$ in Fig. 4, the queue length simply stays at zero. However, this does not affect the packet accumulations at CPU, $p_l(\cdot)$, which in fact should also be decreased at a rate according to the aggregate arrival rate and the bandwidth.

We solve this problem by asking GPU to calculate the quantity of *projected reduction* in queue length as if the fluid queue were not empty during this Runge-Kutta in-

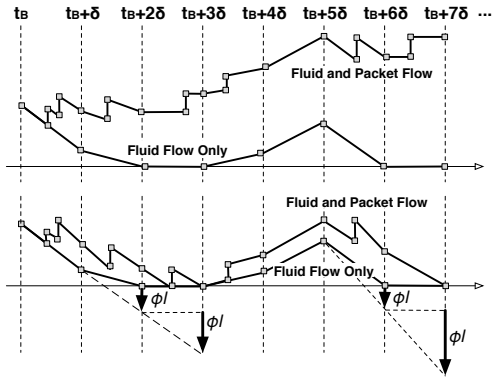

Figure 4: Underflow and fix-up computation.

terval. Suppose the fluid queue l becomes empty or stays empty during the interval between t and $t+\delta$; that is, when $q_l(t+\delta) = 0$. We can calculate the projected reduction in queue length for this interval as:

$$\phi_l(t+\delta) = \max\{(C_l - \Lambda_l(t))\delta - q_l(t), 0\} \tag{12}$$

We can then obtain the cumulative projected reduction in queue length from the beginning of the batch for each Runge-Kutta step of the batch:

$$\Phi_l(t_B + i\delta) = \sum_{k=1}^{i} \phi_l(t_B + k\delta) \tag{13}$$

where $i = 1, 2, \cdots, B\tau$.

On CPU, at the beginning of each batch run, we set $p_l(t_B)$ to zero and set t_l^p to t_B. t_l^p is a variable which we use to record the time at which p_l is last updated. Suppose a packet arrives at the network queue l at time x. We can adjust the current packet queue length p_l:

$$p_l(x) = \max\{p_l(t_l^p) - \Phi_l(x) + \Phi_l(t_l^p), 0\} \tag{14}$$

where$\Phi_{\ l}(t)$ (for an arbitrary time t) can be obtained using a linear interpolation of the values at the Runge-Kutta step boundaries, $t_B + i\delta$, where $i = 1, 2, \cdots, B\tau$. After that, we update t_l^p to be x, and we can now correctly schedule the corresponding packet departure event using Eqn. (11).

4.4 On-Demand Prefetching

Fig. 3 shows that the CPU slave thread needs to wait for the invocation of the three GPU kernel functions to complete and then copy the fluid queue values (including the fluid queue size $q_l(\cdot)$, the packet drop probability $\rho_l(\cdot)$, and the cumulative projected reduction in queue size$\Phi_{\ l}(\cdot)$) from GPU to CPU (shown as step 3). Such interaction happens once every τ number of Runge-Kutta time steps. In low packet traffic situations when the packets arrive sparingly, maintaining this level of interaction between CPU and GPU, however, may not be necessary. For better efficiency, CPU needs to be able to determine *on demand* whether to synchronize with GPU, and if so, what data needs to be transferred from GPU to CPU.

Let $\mathsf{T_{CPU}}$ be the time of the fluid values required by the CPU main thread to process a packet arrival at time t. We set $\mathsf{T_{CPU}}$ to be the end time of its current Runge-Kutta interval:

$$\mathsf{T_{CPU}} = t_B + \lceil (t - t_B)/\delta \rceil \delta \tag{15}$$

Similarly, we use $\mathsf{T_{GPU}}$ to indicate the time up to which GPU has progressed. If the CPU main thread is processing a packet arrival, it needs to determine whether GPU has the needed fluid values ready to be used. If $\mathsf{T_{GPU}}$ is less than $\mathsf{T_{CPU}}$, it means the main thread must wait for GPU to catch up. We accomplish this by using a conditional variable to synchronize between the CPU main thread and slave thread, which is overseeing the progress of GPU. We let the CPU main thread block on the conditional variable if GPU is lagging behind.

The CPU slave thread can determine whether or not to synchronize with GPU between the kernel invocations by choosing whether or not to wait for the completion of all previous invoked kernels. In our implementation the CPU slave thread can invoke the three GPU kernels consecutively at most B times (that's the entire batch) before it is forced to wait for all of them to complete. When GPU is ahead of CPU, the CPU main thread can copy the fluid values from $\mathsf{T_{CPU}}$ to $\mathsf{T_{GPU}}$ in one swoop. Prefetching data from GPU can reduce the frequency of such data transfers and thus improves efficiency. To facilitate that, we use a variable t_{up} to indicate the time of the fluid variables that have already been copied to the CPU main memory.

The complete algorithm, including all optimization techniques mentioned in this and previous sections, is summarized in Alg. 2 and Alg. 3. Alg. 2 describes logic of the CPU main thread and Alg. 3 describes the logic of the CPU slave thread that manages GPU computation.

5. EXPERIMENTS

We implemented the GPU-assisted hybrid network traffic model in our network simulator PRIME [31]. The simulator is designed for parallel and distributed simulation of large-scale networks. For now, the implementation of the hybrid model allows for only sequential execution. We conducted several experiments to validate the hybrid model and assess its performance.

The machine we used for the experiments is a Linux workstation equipped with an Intel i5-750 2.66 GHz CPU, 5 GB memory, and an NVIDIA GeForce GTX 260+ graphics card. We use GCC and CUDA compilers to compile the program with optimization level 2.

5.1 Validation Results

To demonstrate the correctness of the GPU-assisted hybrid model, we use a small network model, which was designed originally by Gu et al. to evaluate their hybrid model [6]. The network topology, as shown in Fig. 5, consists of only 12 nodes and 11 links. The delay and bandwidth of all links are set to be 10 ms and 100 Mbps, respectively. There are 22 RED queues in this example. We place four flows on the network: flow 0 and flow 1 each has 10 long-lasting TCP sessions, all starting at time 0; flow 2 has 20 long-lasting TCP sessions, also starting at time 0; flow 3 has 40 short-lived TCP sessions, starting at 30 seconds and lasting for only 30 seconds. We set the maximum queue length to be 5 MB for all network queues. All TCP sessions in the experiments are assumed to be TCP Reno with a maximum window size of 128 KB.

In the first validation test, we set all flows except flow 2 to be the background fluid traffic calculated on GPU. For flow 2, we select 0, 10, or 20 out of the 20 TCP sessions as the foreground packet-oriented traffic and simulate them on

Algorithm 2 Async hybrid simulation: CPU main thread

$q_l, p_l, t_l^p, \Phi_l \Leftarrow 0$ for all network queues
REQ $\Leftarrow$ false // whether CPU is requesting data from GPU
create and run CPU slave thread
schedule a Runge-Kutta batch event at time 0
$\cdots$ // initialize other simulation variables
While (simulation is not finished) **Do**
  e $\Leftarrow$ eventlist.getEarliestEvent()
  $t_c \Leftarrow$ e.time // current simulation time
  If (e is a Runge-Kutta batch event) **Then**
    If ($t_c > 0$) **Then**
      wait for signal from slave thread previous batch has completed
      copy q_l, ρ_l, and Φ_l for all network queues GPU⇒CPU
    End If
    $T_{GPU}, t_{up}, t_l^p, t_B \Leftarrow t_c$; // start time of the batch
    $T_{CPU} \Leftarrow T_{GPU} + \delta$
    update p_l for all network queues using Eqn. (14)
    $q_l \Leftarrow q_l + p_l$; $p_l \Leftarrow 0$ for all network queues
    signal slave thread to start a new batch
    schedule a Runge-Kutta batch event at time $t_c + B\tau\delta$
  Else If (e is a packet arrival event) **Then**
    $T_{CPU} \Leftarrow t_B + \lceil (t_c - t_B)/\delta \rceil \delta$ // round up
    If ($t_{up} < T_{CPU}$) **Then**
      If ($T_{CPU} > T_{GPU}$) **Then**
        REQ $\Leftarrow$ true // main thread is requesting GPU progress
        wait for signal from slave thread when $T_{GPU} \geq T_{CPU}$
      End If
      copy q_l, ρ_l, and Φ_l for $T_{CPU} \leq t \leq T_{GPU}$ GPU⇒CPU
      $t_{up} \Leftarrow T_{GPU}$ // time of prefetching
    End If
    update p_l using Eqn. (14), $t_l^p \Leftarrow t_B$
    $u \Leftarrow$ uniform(0,1)
    If ($u < \rho_l$ OR $q_l + p_l$ + packet.size $> Q_l^{max}$) **Then**
      drop the packet
    Else
      insert packet into the queue
      $p_l \Leftarrow p_l +$ packet.size
      schedule packet departure event at time $t_c + (q_l + p_l)/C_l$
    End If
  Else
    $\cdots$ // process other simulation events
  End If
End While

Algorithm 3 Async hybrid simulation: CPU slave thread

While (**True**) **Do**
  wait for signal from main thread to start a new batch
  copy q_l for all network queuesCPU⇒GPU
  $i \Leftarrow 0$
  While ($i < B$) **Do**
    invoke `update_flow`, `update_queue` and `update_hop`
    $i \Leftarrow i + 1$
    If (REQ = true AND $t_B + i\tau\delta \geq T_{CPU}$) **Then**
      wait for all previously invoked kernels to complete
      $T_{GPU} \Leftarrow t_B + i\tau\delta$
      REQ $\Leftarrow$ false
      signal CPU main thread $T_{GPU} \geq T_{CPU}$
    End If
  End While
  wait for all previously invoked kernels to complete
  signal CPU main thread the batch has completed
End While

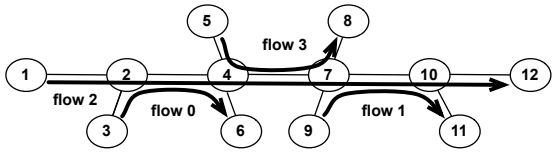

Figure 5: A small network topology with 4 flows.

CPU using the detailed TCP implementation in the simulator. The rest of flow 2 are fluid flows and set as part of the background traffic calculated on GPU. In the case of 0 TCP sessions, we have a pure fluid model, which we use as the baseline for comparison.

We fix the Runge-Kutta step size to be 0.5 ms. Therefore, the lookahead τ is 20 (because the minimum link delay is 10 ms and the Runge Kutta step size δ is 0.5 ms). We start with the batch size $B = 1$. Fig. 6 shows the length of the network queue in node 4 (in the network interface connecting to node 7). We see that the queue length increases rapidly at 30 seconds when flow 3 (with 40 TCP sessions) enters the network causing congestion at the link between node 4 and node 7. The TCP sessions in flow 3 end at 60 seconds, at which time the congestion is relieved and the fluid queue length comes back zero. The results are similar with different mixture of packet and fluid flows. As expected, with more packet flows, the queue length exhibits larger variations since the pure fluid model can only capture average traffic behavior. We observed the same result from the CPU-only model.

In the next validation test, we study the effect of different batch sizes on accuracy. As we mentioned earlier, having a large batch can introduce errors, because CPU can only push the information about the effect of the packet flows (as updated queue length) to GPU at the start of a batch. That is, the influence of packet flows on fluid flows can be delayed if the batch size is large; the result is that the background fluid flows can become less responsive to the changes in the foreground packet flows.

To study this effect, we use the same network model as in the previous experiment. We designate 10 TCP sessions in flow 2 to be packet flows. All other flows are modeled as fluid flows. We compare the results with various batch sizes. Fig. 7 shows the TCP congestion window size of the four flows changes over time for three cases: $B = 1, \tau = 1$; $B = 1, \tau = 20$; and $B = 20, \tau = 20$. Note that the lookahead τ can be set as large as 20; however, we include the case for $B = 1, \tau = 1$ as the baseline for comparison.

We observe similar results for the three test cases. Flow 0 and flow 1 have very similar congestion window trajectories. In comparison, flow 2 has a smaller congestion window because it has a longer round-trip time. At 30 seconds, flow 3 (with 40 TCP sessions) arrives and immediately causes congestion at the link between node 4 and node 7, as shown in Fig. 6. Because of the congestion, flow 2 reduces its congestion window size during this period. Consequently, both flow 0 and flow 1 increase their window size to reclaim the bandwidth handed out by flow 2. The $B = 20$ case shows slightly larger variations than the other two cases. If we keep increasing B more than 20, the delay effect becomes evident and the results become unrecognizable (therefore, it's not shown). This experiment tells us that we cannot arbitrarily increase the batch size; however, we do not yet know exactly

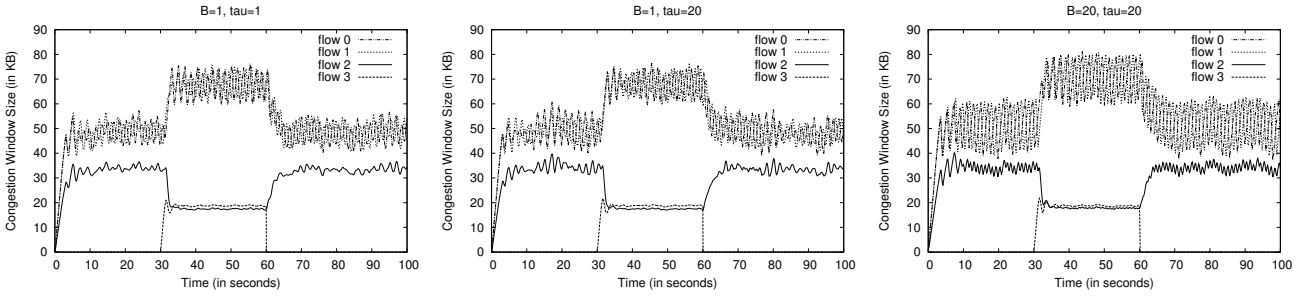

Figure 7: TCP window trajectory under different batch sizes.

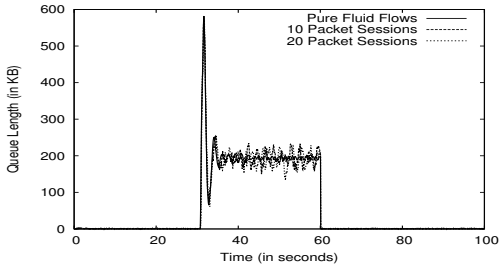

Figure 6: Result from different traffic mixtures.

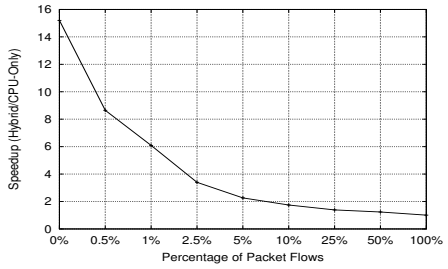

Figure 9: Speedup decreases with more packet flows.

what would be the largest batch size one can choose to get reasonable results.

5.2 Performance Experiments

Next we focus on evaluating the performance of our GPU-assisted hybrid network traffic model. For this experiment, we use the standard campus network model, which has been used for benchmarking the performance of various network simulators. The network consists of a variable number of stub networks, call "campuses". At the top level, the campuses are connected as a ring with additional shortcuts between the campuses that are far apart. Each campus has 504 end hosts, organized into 12 local area networks (LANs) connected by 18 routers. Each campus also has a server cluster with 4 end hosts that can be used as the traffic source. Each LAN consists of a gateway router connecting to 42 end hosts with 10 Mb/s bandwidth and 1 ms delay. A campus is divided into 4 OSPF areas. The links between the routers in the OSPF backbone area and those in the server cluster are configured with 1 Gb/s bandwidth and 10 ms delay. All other router links have 100 Mb/s bandwidth and 10 ms delay.

For the experiment, we choose to simulate 4, 8, and 16 campuses. We cannot run 32 campuses because of the memory limitation on the machine. Traffic on the campus network is generated randomly by the end hosts requesting data from a server in the server cluster either at the same campus or on another campus. Each end host generates 10 TCP flows. Therefore, each campus gets 5,040 traffic flows; for 16 campuses, that's more than 80,000 flows. On average, 50% of the flows are set between campuses. In separate tests, we designate 0%, 0.5%, and 1% of these flows as foreground packet flows simulated using detailed TCP implementation; the rest of the flows are treated as background fluid flows. In this experiment, we set the Runge-Kutta step size to be 0.5 ms and the batch size to be 3.

We compared the performance of our GPU-assisted hybrid model against the original CPU-only implementation. The results are shown in Fig. 8. The normalized execution time is the run time of the model divided by the simulation time. That is, if the normalized execution time is bigger than 1, we have a slow-down: the simulation runs slower than real time. Otherwise, if it's smaller than 1, the simulation is running faster than real time. The CPU-only implementation (left plot) has a slow-down factor ranging from 1.3 for 4 campuses with 0% packet flows to 9.3 for 16 campuses with 1% packet flows. In comparison, our GPU-assisted hybrid model (middle plot) is running faster than real time, except for the case of 16 campuses with 1% packet flows. The right plot in Fig. 8 shows the speedup of the GPU-accelerated model over the CPU-only implementation. With more campuses, more speedup is achieved as the model is benefiting from the more data parallelism available on GPU. As expected, the highest speedup (25x) is achieved by the pure fluid model (i.e., with 0% packet flows).

With increasing packet flows, the speedup decreases. Fig. 9 shows the speedup of our GPU-accelerated model over the CPU-only implementation as we vary the portion of packet flows (for 8 campuses). With more packet flows, the CPU computation is taking over; also, the communication is becoming more expensive as more data needs to be transfered between CPU and GPU. Eventually, the speedup becomes 1 when all flows are packet flows.

Our final experiment looks at the effect of batch size on performance. Fig. 10 shows that the normalized execution time of our GPU-assisted hybrid model decreases as we increase the batch size from 1 to 7. We observe that increasing batch size has a diminishing return in terms of performance improvement. The results show that the performance levels off after the batch size reaches 3 or 4. Beyond that, having a larger batch size does not produce much better perfor-

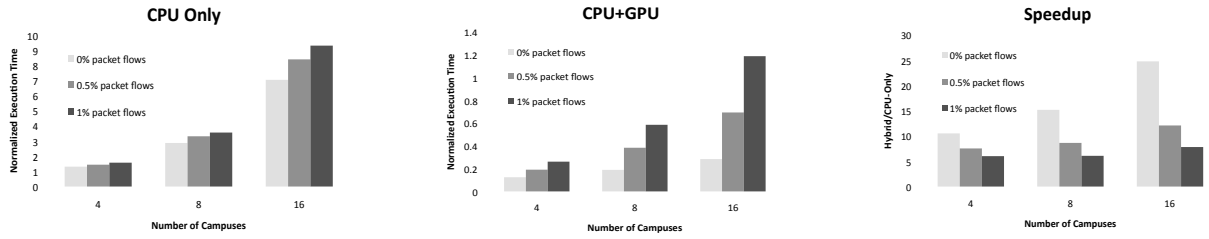

Figure 8: Performance comparison between CPU-only and GPU-accelerated implementations.

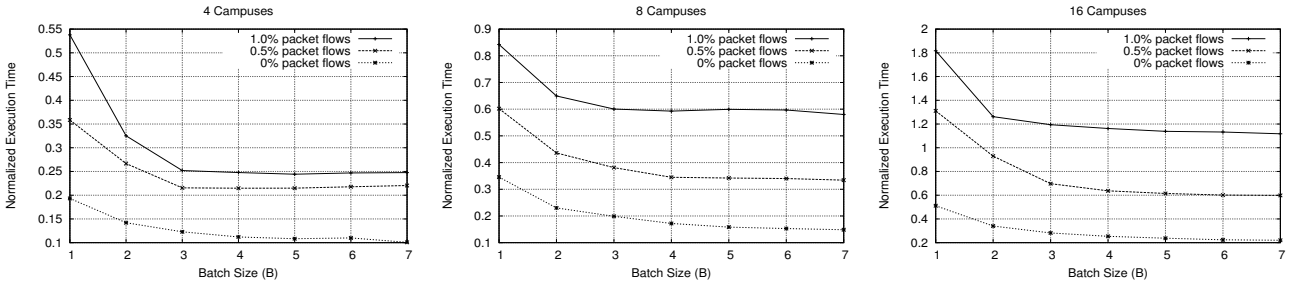

Figure 10: GPU performance decreases with increasing batch size.

mance; in doing so, however, may cause larger errors that can jeopardize the validity of the simulation results.

6. CONCLUSION

In this paper, we propose a GPU-assisted hybrid network traffic model which offloads the numerically intensive background traffic calculations to GPU, and keeps the discrete-event simulation of the foreground packet-oriented transactions on CPU. A novel mechanism that integrates fluid-based and packet-oriented network traffic is introduced, with several optimization techniques that can effectively overlaps CPU and GPU computations and minimize the effect of the inherent communication latencies between CPU and GPU. Experiments show that our method can achieve significant speedup over the CPU-only approach, while still maintaining desirable accuracy.

Our immediate future work includes comparison of the performance impact among the various optimization techniques and further investigation of the loss of accuracy introduced by batch runs. We would like to develop a method for determining the batch size given simulation scenarios. Our current implementation of the hybrid model is sequential. To parallelize the GPU model, we recognize the same lookahead inherent to the fluid equations. However, in order to achieve better parallelism, necessary mechanisms need to be in place to support batch runs. Together with the parallel packet-oriented network simulation, the GPU model shall be able to support massive-scale network simulations with realistic traffic characterization on today's hybrid supercomputing platforms with GPUs.

Acknowledgment

We thank Dr. Kalyan Perumalla at Oak Ridge National Laboratory for the initial discussion of the GPU design. We also thank the anonymous reviewers for their constructive comments. This research is supported in part by the United States National Science Foundation grants (CNS-0836408, CCF-0937964, HRD-0833093), a subcontract from the GENI Project Office at Raytheon BBN Technologies (CNS-0714770, CNS-1346688), and by the National Natural Science Foundation of China (No. 61272087, No. 61363019, No. 61073008 and No. 60773148), Beijing Natural Science Foundation (No. 4082016 and No. 4122039).

7. REFERENCES

[1] B. G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient simulation of agent-based models on multi-GPU and multi-core clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10)*, 2010.

[2] J. S. Ahn and P. B. Danzig. Packet network simulation: Speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking (TON)*, 4(5):743–757, October 1996.

[3] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global Internet. *Computing in Science and Engineering*, 1(1):42–50, 1999.

[4] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[5] GPGPU. General-purpose computation using graphics hardware. `http://www.gpgpu.org/`.

[6] Y. Gu, Y. Liu, and D. Towsley. On integrating fluid models with packet simulation. *INFOCOM*, 2004.

[7] Y. Guo, W. Gong, and D. Towsley. Time-stepped hybrid simulation (TSHS) for large scale networks. *INFOCOM*, pages 441–450, 2000.

[8] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, 2010.

[9] C. Kiddle, R. Simmonds, C. Williamson, and B. Unger. Hybrid packet/fluid flow network simulation. In *Proceedings of PADS'03*, pages 143–152, 2003.

[10] T. Li, N. V. Vorst, and J. Liu. A rate-based TCP traffic model to accelerate network simulation. *Transactions of the Society for Modeling and Simulation International*, 89, 2013.

[11] X. Li, W. Cai, and S. J. Turner. Gpu accelerated three-stage execution model for event-parallel simulation. In *Proceedings of PADS '13*, pages 57–66, 2013.

[12] M. Liljenstam, J. Liu, and D. M. Nicol. Development of an internet backbone topology for large-scale network simulations. In *Proceedings of WSC'03*, 2003.

[13] J. Liu. Packet-level integration of fluid TCP models in real-time network simulation. In *Proceedings of WSC'06*, pages 2162–2169, December 2006.

[14] J. Liu. Parallel simulation of hybrid network traffic models. In *Proceedings of PADS'07*, pages 141–151, June 2007.

[15] J. Liu and Y. Li. On the performance of a hybrid network traffic model. *Simulation Modelling Practice and Theory*, 16(6):656–669, 2008.

[16] Y. Liu, F. L. Presti, V. Misra, D. F. Towsley, and Y. Gu. Scalable fluid models and simulations for large-scale IP networks. *TOMACS*, 14(3):305–324, 2004.

[17] V. Misra, W.-B. Gong, and D. Towsley. Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. *SIGCOMM*, pages 151–160, 2000.

[18] D. M. Nicol. Discrete event fluid modeling of TCP. In *Proceedings of WSC'01*, 2001.

[19] D. M. Nicol, M. Goldsby, and M. Johnson. Fluid-based simulation of communication networks using SSF. In *Proceedings of the 1999 European Simulation Symposium*, 1999.

[20] D. M. Nicol and G. Yan. Discrete event fluid modeling of background TCP traffic. *TOMACS*, 14(3):211–250, July 2004.

[21] NVIDIA. Common Unified Device Architecture (CUDA). `http://developer.nvidia.com/cuda`.

[22] L. Nyland, M. Harris, and J. Prinsn. Fast N-Body Simulation with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics*, pages 21–51, 2005.

[24] H. Park and P. A. Fishwick. A GPU-based application framework supporting fast discrete-event simulation. *Transactions of the Society for Modeling and Simulation International*, 86(10):613–628, 2010.

[25] H. Park and P. A. Fishwick. An analysis of queuing network simulation using GPU-based hardware acceleration. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21(3), 2011.

[26] K. S. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In *Proceedings of PADS'06*, pages 74–81, 2006.

[27] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *Proceedings of the 2008 Spring simulation multiconference (SpringSim'08)*, pages 116–123, 2008.

[28] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. GPU-based real-time execution of vehicular mobility models in large-scale road network scenarios. In *Proceedings of PADS'09*, pages 95–103, 2009.

[29] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Addison-Wesley, 2005.

[30] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, 228:4468–4477, 2009.

[31] PRIME Research Group. Parallel Real-time Immersive network Modeling Environment. `http://www.primessf.net/`.

[32] G. F. Riley. The Georgia Tech network simulator. *MoMeTools*, pages 5–12, 2003.

[33] G. F. Riley, T. M. Jaafar, and R. Fujimoto. Integrated fluid and packet network simulations. In *Proceedings of MASCOTS'02*, pages 511–518, 2002.

[34] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[35] W. Tang and Y. Yao. A GPU-based discrete event simulation kernel. *Transactions of the Society for Modeling and Simulation International*, 89, 2013.

[36] TOP500 Supercomputers Sites. `http://top500.org/`.

[37] M. Verdesca, J. Munro, M. Hoffman, M. Bauer, and D. Manocha. Using graphics processor units to accelerate OneSAF: A case study in technology transition. *JDMS*, 3(3):177–187, 2006.

[38] L. Xu, M. Taufer, S. Collins, and D. G. Vlachos. Parallelization of tau-leap coarse-grained Monte Carlo simulations on GPUs. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–9, 2010.

[39] Z. Xu and R. Bagrodia. GPU-accelerated evaluation platform for high fidelity network modeling. In *Proceedings of the PADS'07*, pages 131–140, 2007.

[40] G. Yaun, D. Bauer, H. Bhutada, C. Carothers, M. Yuksel, and S. Kalyanaraman. Large-scale network simulation techniques: Examples of TCP and OSPF models. *ACM SIGCOMM Computer Communication Review*, 33(3):27–41, 2003.

[41] J. Zhou, Z. Ji, M. Takai, and R. Bagrodia. MAYA: Integrating hybrid network modeling to the physical world. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(2):149–169, 2004.