GPU-Accelerated fluid flow approximation of the Active Queues Management algorithms

ADAM DOMAŃSKI¹, JOANNA DOMAŃSKA², TADEUSZ CZACHÓRSKI²

¹ Institute of Informatics Silesian Technical University Akademicka 16, 44–100 Gliwice, Poland
adamd@polsl.pl

² Institute of Theoretical and Applied Informatics
Polish Academy of Sciences
Baltycka 5, 44–100 Gliwice, Poland
{joanna,tadek}@iitis.gliwice.pl

Received 4 May 2013, Revised 10 June 2013, Accepted 17 July 2013

Abstract. In the article we study a model of TCP connection with Active Queue Management in an intermediate IP router. We use the fluid flow approximation technique to model the interactions between the set of TCP flows and AQM algorithms. We perform the fluid flow approximation in the CUDA environment.

1. Introduction

Design of technology for TCP/IP networks is one of the most important topics in the field of telecommunications networks. The main problem is still the modeling of congestion control mechanisms. The development of new active queue management (AQM) routers allows to increase the performance of Internet applications.

A number of analytical models of AQM in IP routers in open-loop scenario – because of the difficulty in analyzing AQM mathematically – was already presented, [13], [6]. In this article we try to use the nonlinear dynamic model of TCP [10], [19] to analyze the AQM systems. This model enables application of control rules to address the basic feedback nature of AQM.

We use the fluid flow modeling methodology based on mean value analysis. This analytical method of modeling has a great potential in analyzing and understanding various network congestion control algorithms [15]. The models based on fluid flow
approximation are able to capture the dynamics of TCP flows [21] and allow to analyze networks with a large number of flows. Here, we use this method to compare routers having different active queue management principles and transmitting TCP/UDP flows. The model allows to study not only the steady-state behavior of the network, but also the transient behavior when a set of TCP flows start or finish transmission. We concentrate on transient average router queue length for different AQM strategies. In this paper we present results of calculations obtained using the GPU environment. The computation time in the CUDA environment are compared with the time of calculation in standard CPU. For large matrix multiplication we propose to use the CUBLAS module (Basic Linear Algebra Subprograms) prepared by NVIDIA.

The rest of this article is organized as follows. The section 2 describes the fluid flow model of AQM router supporting TCP/UDP flows. The section 3 shows obtained results. The conclusions are presented in section 4.

2. Fluid-flow model for the network case

This section presents a fluid flow model the AQM router supporting TCP/UDP flows.

The model presented in [15] demonstrates TCP protocol dynamics. This model ignores the TCP timeout mechanisms and allows to obtain the average value of key network variables. This model is described by the following nonlinear differential equations [9]:

\[
\frac{dW_i(t)}{dt} = \frac{1}{R_i(t)} - \frac{W_i(t)W_i(t - R(t))}{2R_i(t - R_i(t))} p(t - R_i(t)) \tag{1}
\]

\[
\frac{dq(t)}{dt} = \sum_{i=1}^{n} \frac{W_i(t)}{R_i(t)} - C \tag{2}
\]

where:
- \( W_i \) = expected TCP sending window size (packets) for i-flow
- \( q \) = expected queue length (packets)
- \( R \) = round-trip time = \( q/C + T_p \) (secs)
- \( C \) = link capacity (packets/sec)
- \( T_p \) = propagation delay (secs)
- \( N \) = number of TCP sessions
- \( p \) = packet drop probability
The maximum values of $q$ and $W$ (queue length and congestion window size) depend on the buffer capacity and maximum window size. The dropping probability $p$ depends on the queue algorithm. We do not distinguish explicitly TCP and UDP connections, saying only that UDP connections influence the capacity $C$ of the output link.

For the RED algorithm, dropping probability $p_{RED}$ is growing linearly from 0 to $p_{max}$:

$$p_{RED} = p_{max} \frac{x - Min_{th}}{Max_{th} - Min_{th}}$$  (3)

For the CHOKe algorithm, the probability $p_{CHOKE}$ depends on the number of packets of the i-stream relative to the total buffer occupancy. For simplicity, the model assumes that the number of packets belonging to a single stream in the queue is the same for all streams so. It follows that the probability of packet loss is inversely proportional to the number of the streams.

$$p_{CHOKE} = \frac{1}{N}$$  (4)

An extension of the fluid flow approximation model allows to calculate the transmission parameters for the network of transmission nodes [11]. The extended model assumes that the network $V$ consists of $K$ routers. The queues are represented by vectors $Q$ and $X$. The probability of packet dropping is represented by vector $P(x)$. Matrix $A$ represents the structure of the network $V$. The rows of the matrix correspond to the flows in the network. The columns of the matrix represent individual nodes. If the flow $i$ travels through a node $k$, the element $a_{ik}$ is set to 1, otherwise it is set to 0.

Matrices $A$ and $P(x)$ are used to create a new matrix $AP$ [15]. The rows of matrix $AP$ are obtained by multiplying rows of matrix $A$ with the proper item of vector $P$. This matrix is used to obtain the complete loss probability for packets on the path (for all flows). The row of the matrix $AP$ describes the drop probability for all routers on path. The total packet drop probability is a combination of the individual probabilities for the routers along the path from the source to the destination. An easier solution is to calculate the probability of correct packet transmission from the source to the target. Hence, the dynamics of the TCP window for the network of nodes can be represented as:

$$\frac{dW_i(t)}{dt} = \frac{1}{R_i(t)} - \frac{W_i(t)W_i(t - R(t))}{2R_i(t - R_i(t))}(1 - \prod_{n=0}^{K}(1 - AP(x)_{ni}))$$  (5)
3. Results

The computations were made with the use of PyLab (Python numeric computation environment) [18], a combination of Python, NumPy, SciPy, Matplotlib, and IPython. The graphs shown below present transient system behavior, the time axis is drawn in seconds. The access to Nvidia’s CUDA parallel computation API [22] is obtained using PyCUDA.

The parameters of AQM buffer:
- $M_{in_{th}} = 10$,
- $M_{ax_{th}} = 15$,
- buffer size (measured in packets) = 20,
- weight parameter $\alpha = 0.007$.

The parameters of TCP connection:
- transmission capacity of AQM router: $C = 0.075$,
- propagation delay for $i$-th flow: $T_{pi} = 2$,
- initial congestion window size for $i$-th flow (measured in packets): $W_i = 1$.

The obtained mean queue lengths for TCP connections are presented in table 1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Nb of streams</th>
<th>Nb of packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOKE</td>
<td>1</td>
<td>7.98664081264</td>
</tr>
<tr>
<td>CHOKE</td>
<td>2</td>
<td>8.63146812018</td>
</tr>
<tr>
<td>CHOKE</td>
<td>5</td>
<td>10.0998514529</td>
</tr>
<tr>
<td>CHOKE</td>
<td>10</td>
<td>11.0167546717</td>
</tr>
<tr>
<td>CHOKE</td>
<td>11</td>
<td>11.7731309893</td>
</tr>
<tr>
<td>RED</td>
<td>1</td>
<td>8.57089136683</td>
</tr>
<tr>
<td>RED</td>
<td>2</td>
<td>9.05376778822</td>
</tr>
<tr>
<td>RED</td>
<td>5</td>
<td>10.3805817389</td>
</tr>
<tr>
<td>RED</td>
<td>10</td>
<td>11.1549893996</td>
</tr>
<tr>
<td>RED</td>
<td>11</td>
<td>11.7731309893</td>
</tr>
</tbody>
</table>

Table 1. The obtained mean queue lengths $Q(t)$
Figure 1 shows the queue behavior in case of one TCP flow and the CHOKe queue. The size of congestion window increases until the buffer reaches $M_{th}$. The algorithm drows ”CHOKe victim” and always achieves success. Hence the probability of packet loss is equal to the one. Packets are dropped, the size of congestion window decreases causing a slow decrease of the queue length – this pattern is repeated periodically. The similar situation exists for the two streams (figures 3, 4). The probability of removing a packet is equal to $\frac{1}{2}$. Probability of removing the packet by RED mechanism is in both cases much smaller. Comparing the behavior of the CHOKe algorithm to the RED, it is clear that CHOKe algorithm is better in the case of aggressive (stealing most of the bandwidth) streams.

In our fluid flow aproximation we calculate the dynamic of the window ($\frac{dW}{dt}$) for all TCP/UDP streams and then we calculate the queue occupancy. In the CUDA environment we can perform these calculations in parallel.

A simple program using pucuda is presented below:
Fig. 2. RED queue 1 TCP/UDP flow

# Import modules

import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

# Preparing matrix 4x4 of random numbers:
a = numpy.random.randn(8, 8)
a = a.astype(numpy.float32)

# Memory allocation in GPU
a_gpu = cuda.mem_alloc(a.nbytes)

# Transfer to GPU
cuda.memcpy_htod(a_gpu, a)

# Simple program written in nvcc
gpu_rozkaz=''
    _global_
    void doublify(float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
    '''
#program execution
mod = SourceModule(gpu_rozkaz)
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

#data transfer from gpu to cpu
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a

Listing 1. Simple program using pycuda

The execution of the calculations consists of four steps:

- input data preparation (array of 32-bit float),
- transport data to GPU,
- calculation in GPU,
- downloading data from GPU to CPU.

In the fluid approximation we prepare arrays of $W_i(t)$, $W_i(t-R(t))$, $R_i$, $R_i(t-R(t))$, we send this arrays to GPU and make calculations. Then we download the array of the new $W_i(t)$. 
Figure 5 shows the comparison of calculation time in GPU and CPU environments depending on the number of streams. This time is strongly indetermined. Our experiments were repeated a hundred times, and the graph shows the mean values. At the beginning of the computation in GPU time is relatively large. This is related to the time required to initialize the GPU. In a later stage, the time slightly depends on the number of processed flows.
The fluid flow approximations for a large number of nodes involves the multiplication of large matrices. These calculations can also be performed in a GPU environment. Figure 6 shows times of matrices multiplication (depending on matrices size). The computation time increases very slowly with the increase of the size of the matrix. This calculation was performed using a *single* block of threads. So we can multiplicate only matrices of size 22x22.

![Fig. 6. Times of matrices multiplication in GPU environment](image)

Additional Python bindings to simplify matrix multiplication operations can be found in the program pycublas [23]. CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDATM runtime. It allows access to the computational resources of NVIDIA GPUs. The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary. CUBLAS attaches to a single GPU and does not auto-parallelize across multiple GPUs.

PyCUBLAS multiplies matrices smaller than 65536-by-65536 [23]. The matrix multiplication using pycublas is very simple as presented below:

```python
import numpy
from pycublas import CUBLASMatrix

A = CUBLASMatrix(numpy.mat([[1,2,3],[4,5,6]],numpy.float32))
B = CUBLASMatrix(numpy.mat([[2,3],[4,5],[6,7]],numpy.float32))
C = A*B
```
Fig. 7. Times of matrices multiplication.

Listing 2. Matrix multiplication using pycublas

All CUBLAS alloc and free calls are mapped to the CUBLAS Matrix object’s life in Python, so you don’t have to worry about memory management (other than filling up the card, of course). For matrices multiplication of size (4160x4160) Cublas is 43x faster in comparison to the calculation using the library ”numpy”. Figure 7 shows times of matrices multiplication (using CUBLAS) depends on matrices size.

4. Conclusions

This article confirms the advantage of the CHOKe algorithm over standard RED for aggressive streams. Using the choke algorithm is insignificant in the case of a large number of streams with the similar intensity. In this article we also present how to perform calculations in GPU environment. We used GPU block of thread we calculated dynamic of conguestion window for all streams in one step. We used a *single* block of threads. So we could perform calculation for 600 streams.

For matrix multiplication used in extended model for network of nodes we proposed to use pyCUBLAS.
Acknowledgements

This research was partially financed by Polish Ministry of Science and Higher Education project no. N N516479640

References


Aproksymacja płynna algorytmów AQM – wspomagana przez GPU

Streszczenie

Artykuł opisuje zastosowanie aproksymacji płynnej do modelowania interakcji pomiędzy zbiorem strumieni TCP, a mechanizmami aktywnego zarządzania buforami (AQM). Obliczenia zostały przeprowadzone w środowisku GPU. Wyniki przedstawione w artykule potwierdzają przewagę algorytmu CHOKe nad standardowym algorytmem AQM: mechanizmem RED.