Project Report: On GPU-based Parallel Peer-to-Peer Simulation

Di Niu and Zhengjun Feng

dniu@eecg.toronto.edu, zhengjun.feng@utoronto.ca

1. Introduction

Peer-to-peer (P2P) content distribution systems such as file-sharing (e.g. BitTorrent), live streaming (e.g. PPLive) and video-on-demand (e.g. Joost) systems have become enormously popular on today's Internet. These systems organize tens of thousands or even millioins of participating user hosts into a randomized mesh topology, where each user maintains connection with a subset of other peer users, sharing the desired data. In a typical file-sharing scenario like BitTorrent, a large file is broken down into *k* granularity blocks. The goal is to distribute all the blocks to all the users by letting peers exchange these blocks in a decentralized fashion. To attain this goal, each peer has to decide which neighboring peer to transmit to and which block to transmit at a given time. Distributed block scheduling and neighbor selection protocols in P2P networks have become a very active research area in recent years [1].

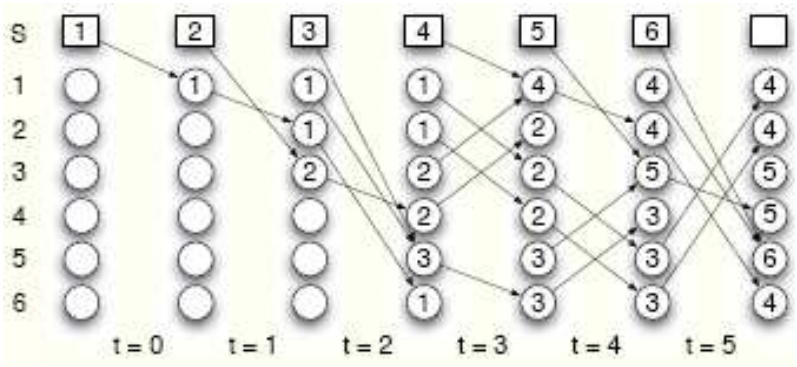

Figure 1: Round-based P2P Simulation

Even-driven simulation (e.g. Ns-2 [9]) is a popular tool to study the behaviors of various networks, such as wireless ad hoc networks. However, such simulation is very slow when applied to P2P networks, as the network size scales to more than tens of thousands. Besides, most well-known event-driven simulators like Ns-2 focus on packet-level traffic characterization, which is only a secondary consideration in P2P simulation. P2P simulation, instead, has a more vital task to understand how the peer selection and data block scheduling algorithms can impact the data dissemination speed in the network.

One alternative to simulate the P2P protocol is the round-based simulation illustrated in *Figure 1*. In each round, each peer independently selects a neighbor and transmits a certain block it has buffered to the receiver by certain protocols. A peer that has all the data blocks is called a *seed*, otherwise, it remains a *downloader*. The goal is to see how many rounds it takes for certain

protocols to broad all *k* blocks to all *N* peers. Such a round-based idea can be combined with event-driven simulation to perform Parallel Even-Driven Simulation (PEDS). All the network events such as block replication can be viewed as events. The time is discretized into time slots. All the events whose timestamps fall into the same time slot can be executed in parallel for this slot. Such simulation can approximate the actual event-driven simulation with a little bit compromise of accuracy. However, due to the parallel processing, its scalability can satisfy the need for modeling real P2P applications, which event-driven simulators like Ns-2 fails to do.

In this project, we explore the feasibility of using GPU to expdite parallel P2P simulation. As a first attempt, we consider the following model of BitTorrent. An undirected network is generated as a random graph, each pair of peers being connected with probability *p*. *k* data blocks are to be broadcast from a single seed to all the other *N-1* peers by letting peers transfer a block to its neighbor in each round. We say a block is useful to a peer, if it has not obtained the block. Two algorithms are implemented: 1) Random Useful Block (RUB), where each peer randomly selects a neighbor as the receiver and transmits a random data block that is useful to the receiver; 2) Global Rarest First (GRF), where each peer selects a random neighbor as the receiver and transmits a useful block that is the rarest within the network.

To optimize the GPU performance, we implement RUB using two different methods, and GRF in three different approaches. We achieved an 8x GPU speedup for RUB and a 25x GPU speedup for GRF as compared to the CPU codes. This strongly supports our conjecture that there is a great potential to leverage the parallel processing ability of GPU to enhance the scalability and performance of P2P simulations.

2. Related Work

Parallel distributed event-driven simulations (PDES) [3,4,5,6] have been developed for many years and some PDES tools have been developed for simulation of large-scale Internet applications. In PDES, instead of having a single event scheduler, there are multiple event schedulers which interact with each other. The core engine of a PDES carries out simulation round by round. In each round, processors/threads synchronize with each other to determine a look-ahead window and process only the events in that window. In such a simulation, the chronological order of the processing of events is guaranteed.

However, most of the current PDES simulation packages including pdns [3], SSF [4], and GloMoSim [6] are designed for packet level simulation and are not able to scale to millions of nodes. Lin. et al. [2, 7] proposed a simulation architecture for P2P systems which was the first to take advantage of the features of P2P systems. It introduces a slow-message relaxation optimization which trades simulation accuracy for speed. Emulation [8] is another mechanism to study large-scale systems. To the best of our knowledge, GPU-based parallel P2P simulator has not been observed in literature. By implementing some basic algorithms in a simple P2P simulation scenario, this course project represents the first step towards GPU-based parallel event-driven simulation for large-scale P2P networks.

3. Algorithms and Methodology with GPU

We first implemented the CPU-based simulation for RUB and GRF from scratch using serialized executions. We then designed and implemented the corresponding algorithms for the GPU-based simulation. In order to optimize the performance of GPU, we tried two different designs for RUB: Peer per Thread without using shared memory and Peer per Thread using shared memory; and three designs for GRF: Peer per Thread, Data Block per Thread and Multi-Data-Block per Thread. We will introduce the details of these schemes in this session.

Before we touch upon the details of the kernels on GPU, it is worth noting that several data structures are maintained in the global memory of GPU:

- Neighbor list: stores the random generated neighbor list for each peer based on probability setting of p ($0 < p <= 1$).
- Data block matrix: contains the original data block information where the first peer is the seed (containing all the data blocks) and all the other peers are downloader with empty data blocks.
- Seed counter: contains the number of seeds which indicates the number of peers that have completed downloading data blocks, and is set to 0 at the beginning of each round.

3.1 RUB

3.1.1 Not Using Shared Memory

This is the naive design of RUB using global memory for all the data storage. Within each round, the kernel generates a random number and selects a random target peer based on the number generated, compares and counts the difference of the data blocks with the target peer, generate another random number and compare the data block matrix again to get the data block index for transmission, finally transmit the indexed data block to target peer.

The drawback of this naïve design is that the data block matrix is compared twice. The first compare is to count the number of difference, and the second compare is to get the index of the different data block. In addition, since the target peer is randomly selected, it is very possible that two or more peers choose the same target peer, and hence the additional compare may cause more conflicts between threads (peers). So in order to remove the redundant data matrix compare, we designed the shared memory based RUB algorithm.

3.1.2 Using Shared Memory

Since the read and write operation on shared memory is much faster than global memory, the shared memory can be utilized to store the indexes of different data blocks during the comparison mentioned in 3.1.1 and remove the second data comparison. On the other hand, since the amount of shared memory is limited to 16K bytes per GPU block, the memory available for each thread is limited, so we can only store a limited number of indexes.

Shared memory can provide fast data access like read and write operation, but we need to avoid the

bank conflicts between the threads. Table 1 shows how the shared memory is allocated per thread in order to avoid the bank conflict. Take the GPU block size of 64 as an example, the available maximum integers that each thread can store is 16KB/64/4 = 64, but if we allocate memory size of 64 integers for each thread, all the 16 threads (half warp) will start the data access from bank 0 and cause bank conflict. So in order to avoid this conflict, we deduce the memory size of one integer for each thread, and can get all the 16 threads start at different bank as shown in *Figure 2*. Thread 0 accesses bank 0, thread 1 access bank 15, thread 2 access bank 14, and so on.

Shared Memory Size (KB)	Block Size (# of threads)	Available Max Integers/threads	Designed Integers/Thread
16	32	128	127
16	64	64	63

Table 1: shared memory data size to avoid the bank conflicts

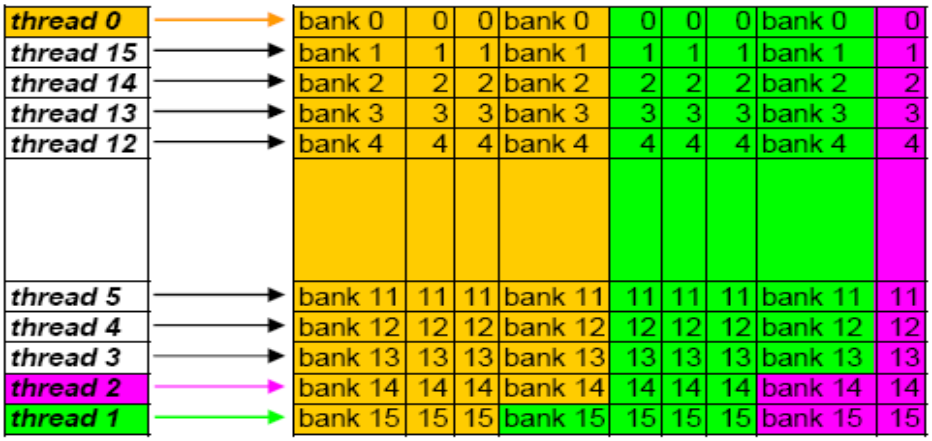

Figure 2: bank access on shared memory to avoid bank conflicts

With limited shared memory the algorithm will take a bit more rounds to finish the data transmission, however the evaluation in 5.1 shows that this design performs better than the previous design and has better scalability.

3.2 GRF

Two designs for GRF are implemented on GPU: one is based on peer per thread, which means each thread handles a peer; another design is to let each thread handle one or multiple data blocks in each peer's buffer.

3.2.1 Peer per Thread

In the Peer per Thread scheme for implementing GRF on GPU, we let each peer handle all the actions of each sender peer in each round. For GRF, we maintain a Counter table in global memory to indicate how many copies of each data block there are in the network. Similar to RUB, each thread, as a representative of each sender, first selects a random peer from its neighbor list and then scans both the buffers of the sender and receiver sequentially. If it finds a data block that exists in the sender but is vacant in the receiver, it checks the Counter of this data block. If this data block

has fewer copies than any of the blocks scanned so far, then it is deemed as the rarest block. The sender sends the rarest block to the receiver when the scan ends. In the kernel for the above operations, there are as many threads as peers.

The Counter table needs to be updated at the end of each round. As two peers might be transmitting the same data block to the same receiver, if these two sender threads update Counter in parallel, the Counter of the transmitted data block will be increased twice. To avoid this, we record all the peer buffer positions that need to be updated in each round, and implement another kernel for Counter update. In the kernel for Counter update, there are as many threads as the total number of buffer positions (which is equal to # of Peers * # of Data Blocks).

3.2.1 Data Block per Thread

In order to build in more processing parallelization, we propose Data Block per Thread that reduces the amount of work that each thread does while creating much more parallel threads for processing. Since the main bottleneck lies in buffer comparison, to expedite this, we let each *thread block* be responsible for one sender peer, while letting each thread in this *thread block* compare one buffer position in this peer with the corresponding buffer position in its receiver. In the case when the number of data blocks is less than 512, there can be one thread handling the comparison of exactly one buffer position, instead of *k* positions, and thus greatly improves performance.

We only use one thread (say thread 0) in each *thread block* to select receiver at the beginning and update the receiver buffer at the end. We maintain the selected receiver ID, the block index to transmit (*blk2Send*), and the Counter of the rarest block (which is useful to the receiver), *min*, in the shared memory of the sender peer. When it comes to the parallel comparison of all the buffer positions between the send and receiver, intuitively, the following algorithm should be used:

Do in parallel for each data block *blkIndex :*
If *blkIndex* is in the sender **and** not in the receiver
    Do in one atomic operation:
        If Counter[blkIndex] < min
            Do
                min = Counter[blkIndex]
                blk2Send = blkIndex

However, it is non-trivial to ensure the atomic operation in the above, without incurring error or much performance degradation. In fact, we have tried the lock-based solution, which turns out to be extremely slow. To solve this dilemma, we use the following algorithm:

Do in parallel for each data block *blkIndex :*
If *blkIndex* is in the sender **and** not in the receiver
        Do atomicMin(&min, Counter[blkIndex]);
Synchronize
Do in parallel for each data block *blkIndex :*
If Counter[blkIndex] == min
    If *blkIndex* is in the sender **and** not in the receiver

Do atomicExch(&blk2Send, blkIndex);

By doing this, we solve the difficulty of doing one big atomic operation by splitting it into two smaller atomic operations which could be achieved by the built-in functions in CUDA, namely, atomicMin() and atomicExch(). In this implementation of the kernel for receiver and block selection, there are as many *thread blocks* as peers. The kernel for counter updating is the similar to the one in Peer per Thread.

3.2.2 Multi-Data-Block per Thread

We now extend the Block per Thread scheme to the cases when the total number of data blocks is more than 512. In this situation, we still view each *thread block* as a sender peer. Receiver selection, buffer and counter updating are performed in the same way as in Section 3.2.2. The only difference here lies in the buffer comparison. Instead of letting each thread in the sender deal with the comparison of only one buffer position, we let each thread handle the comparisons of multiple positions. Specifically, buffer position i is handled by thread with index (i mod thread block size). For example, if there are 1024 data blocks (buffer positions), and the thread block is of size 512, then thread one handles buffer position 1 and buffer position 513. Except that synchronization should be dealt with more carefully, most parts of implementing Block per Thread for an arbitrary number of data blocks are similar to the case with less than 512 data blocks.

4. Evaluation

We compare the performance of our GPU-based parallel simulation algorithms with their corresponding serial implementations using CPU. We define the processing ability of GPU (CPU) to be the number of rounds of simulation that GPU (CPU) can complete per second. We vary the number of peers and the number of data blocks to be disseminated to see how the performance speedup of GPU over CPU changes. We consider random graphs with connection probability p between two peers being 0.5 in all the experiments. For each set of parameters, we run both the GPU and CPU code for 10 times and take the average.

4.1 RUB Algorithm

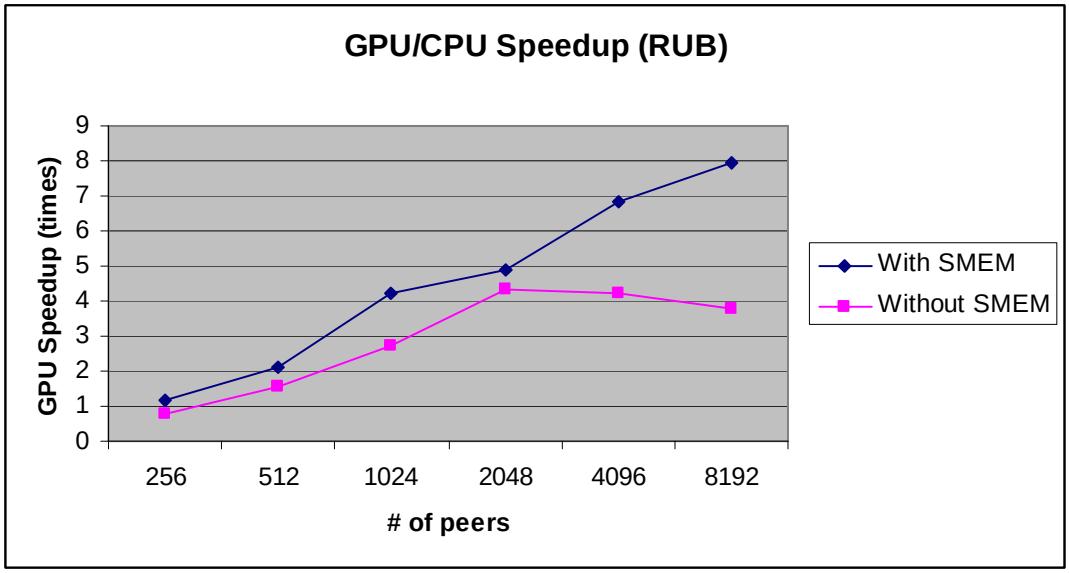

Figure 3: GPU to CPU speedup (# of data blocks = 256)

Figure 3 shows the GPU speedup for RUB over CPU with an increasing number of peers when the number of data blocks is fixed to 256. Both the design with and without shared memory are shown in the figure. As we can see, RUB with shared memory can get a speedup of 8x but only 4.3x without using shared memory (the naive design). In addition, the design with shared memory has better scalability with the increase of number of peers due to fast data access of shared memory.

The speedup of the naive design decreases after the number of peers exceeded 2048. The main reason of this is because with random receiver selection, there could be a larger number of senders trying to transmitting to the same receivers as peer number increases. A very conservative estimation from the theory is that for a network of *N* peers, the maximum number of senders a receiver can have is at least ln(N)/lnln(N) with probability at least 1-1/N. Hence, the competition for accessing memory resources will become a bottleneck. This effect is exacerbated by the fact that the naive design compares each peer's buffer with that of its randomly selected receiver twice.

For the design with shared memory, it also encounter the resource competition, however, the limited size of shared memory also limited the number of data comparisons between the sender and the receiver. In addition, it only compares the buffer difference once, and the random block index is read from the shared memory instead of comparing the matrix again. So the resource competition is greatly reduced with the design of shared memory.

Figure 4 shows the GPU to CPU speedup with increasing number of data blocks when the number of peers is fixed to 1024. In general, the design using shared memory performs better than the one without using shared memory. The trends for both designs are similar, with the number of data blocks increasing, the GPU speedup over CPU is slowed down. This is mainly due to the increased workload for each thread, i.e., more buffer positions need to be compared. We observe that the GPU performs much better when letting each thread do less and simple things instead of tedious and intelligent tasks. The Data Block per Thread scheme for GRF in Section 4.2 is an example of this observation. For the RUB design using shared memory, with the number of data blocks

increasing, it takes more rounds (as shown in figure 5) to finish the P2P simulation due to the limited size of shared memory, which reduces the randomization of data block selection. However, in Figure 5, we see the inaccuracy is bounded to a small value.

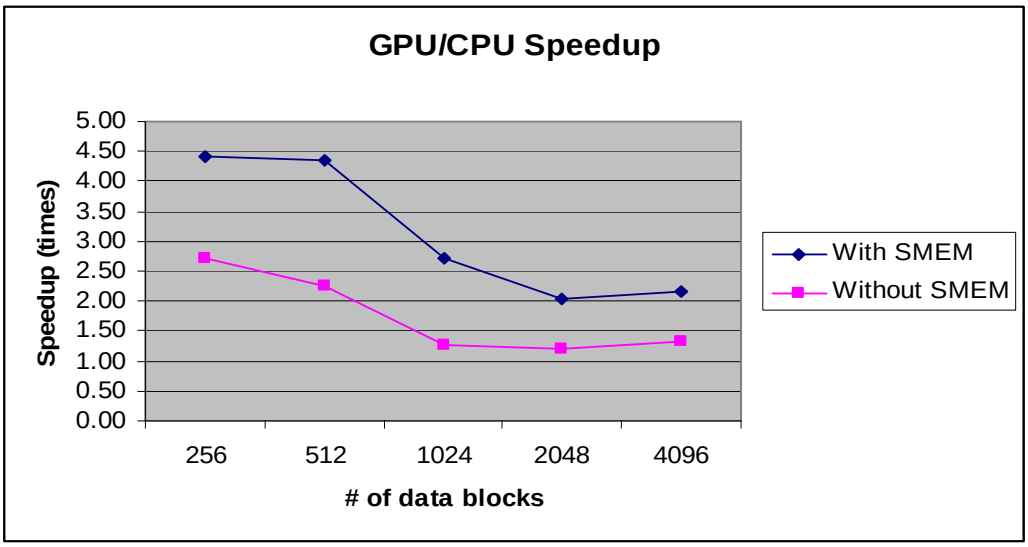

Figure 4: GPU to CPU speedup (# of peers = 1024)

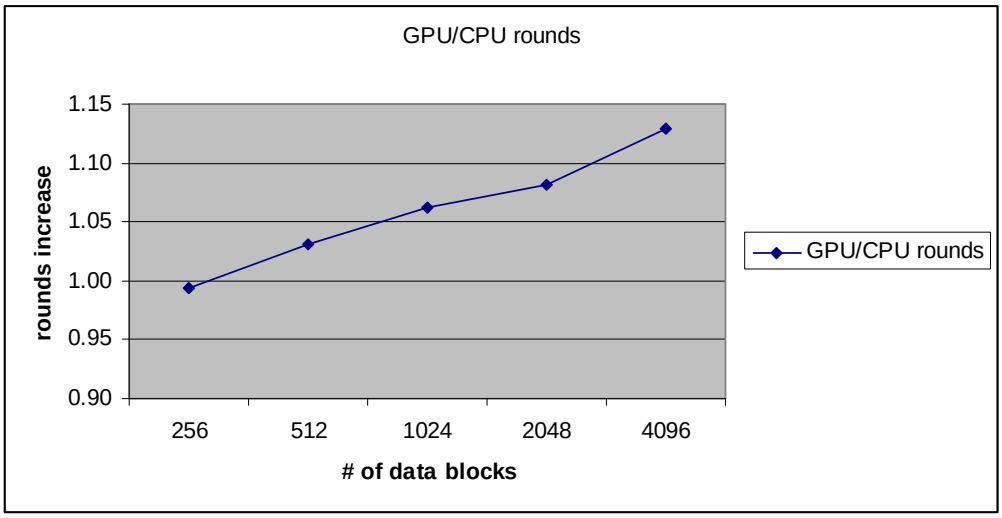

Figure 5: The accuracy of GPU simulation of RUB with shared memory. Evaluated by the ratio between GPU results (rounds) and CPU results (rounds). Number of peers = 1024

4.2 GRF Algorithm

Figure 6 and Figure 7 show the speedup of Data block per Thread and Peer per Thread over the serial CPU implementation as the number of data blocks grows. From Figure 6, we can see that for Peer per Thread, the benefit of GPU goes down after the number of data blocks exceeds 512. This

is because each thread is doing too much buffer scanning in each round, degrading the performance. In contrast, with parallelization at the data block level, Data Block per Thread does not have a bottleneck when processing buffer comparison; its speedup against CPU increases up to 21x, when 512 data blocks are to be distributed to 16384 peers.

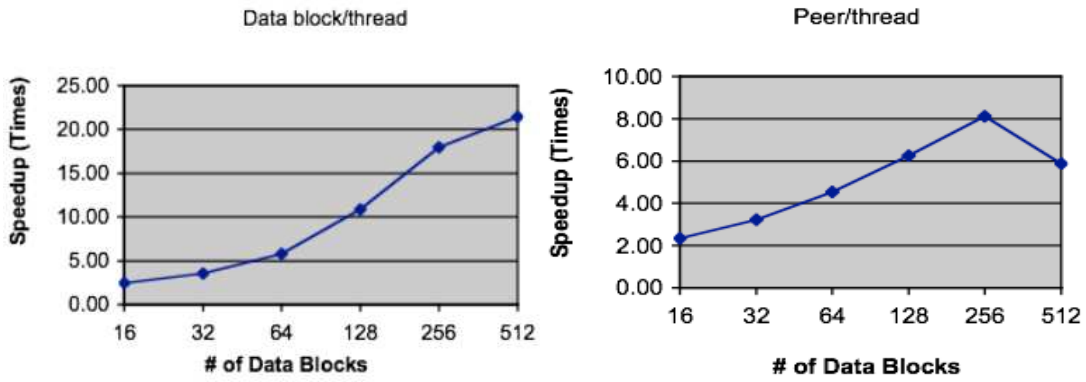

Figure 6: Speedup Comparison of Data Block per Thread v.s. Peer perThread

Number of Peers = 16384. For one thread per block , grid size = 16384 , block size = # data blocks. For one thread per peer , grid size = # of Peers/512 , block size = 512

From Figure 7, we can take a closer look at the speedups of Multi-data-block per Thread v.s. Peer per Thread as the number of data blocks grows beyond 512. We see that as each thread does more and more buffer scanning jobs in Peer per Thread, the benefit of GPU is diminishing to around 4x when the number of data blocks increases to 2048. In sharp contrast, the speedup of Multi-data-block per Thread increases to 25x.

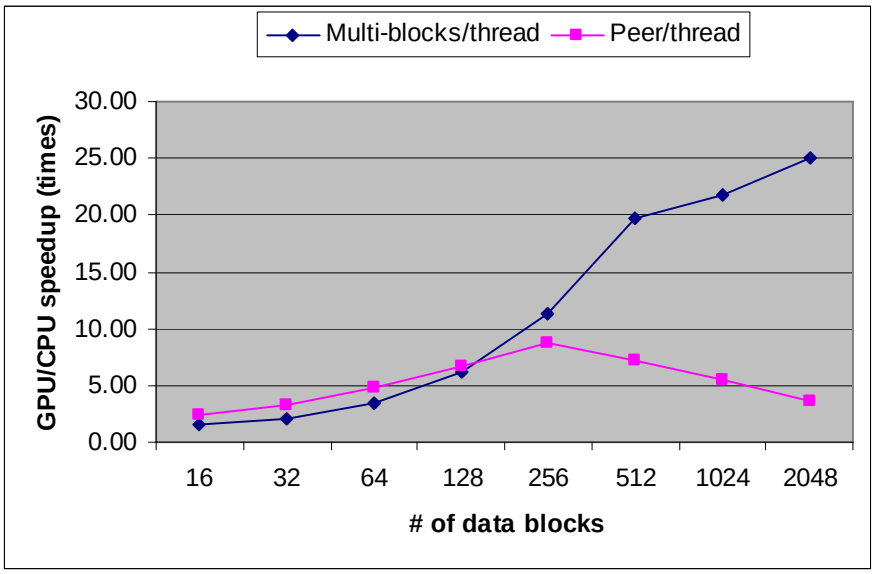

Figure 7: Multi-data-blocks per Thread v.s. Peer per Thread (# of peers = 8192)

From Figure 8 we can see that as the number of peers increases, the speedup of Peer per Thread

slows down. Although each peer still processes the same amount of buffer scanning (there are 512 data blocks in total), yet with random receiver selection, there could be a larger number of senders trying to transmitting to the same receivers. In this case, the senders all need to compare their buffers with the receiver's buffer, causing much competition in accessing memory resources and thus performance degradation. Again, the performance speedup of Data Block per Thread is increasing as the number of peers increases, with a maximum value of 21x in our simulation for 512 data blocks and 16384 peers.

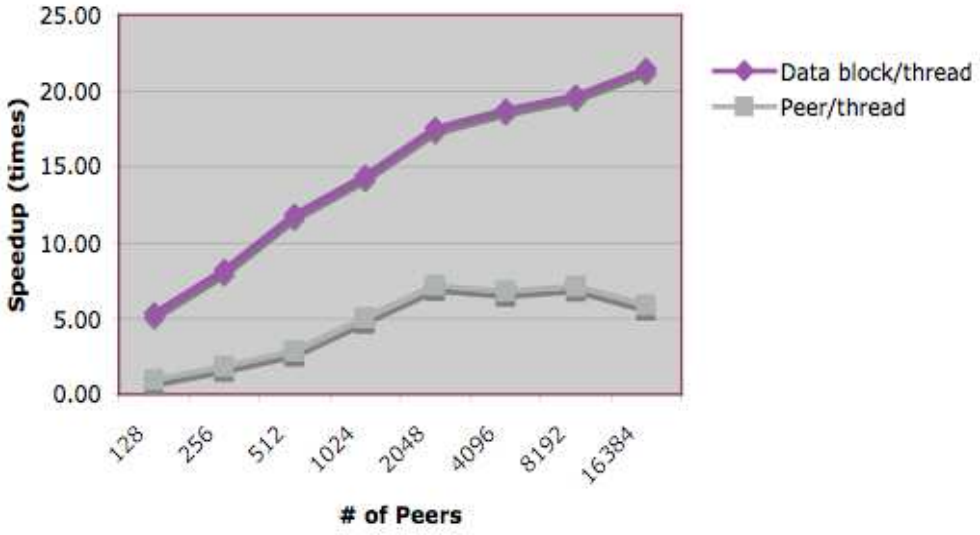

Figure 8: GPU to CPU speedup (# of data blocks = 512)

For one thread per block , grid size = # of Peers , block size = 512

For one thread per peer , grid size = # of Peers/512 , block size = 512

5. Conclusions & Future Work

We simulated two P2P content distribution algorithms, namely, Random Useful Blocks (RUB) and Global Rarest First (GRF) using GPU. Using serial CPU code as a benchmark, GPU can speedup RUB simulation up to 8x with a small error when shared memory is used. It can speedup RUB up to 4.3x with accurate results when shared memory is not used. For GRF, we can get up to 25x speedup using Multi-data-block per Thread and up to 8x speedup using a Peer per Thread scheme.

In this project, we preliminarily explored the feasibility of using GPU to implement parallel simulation architectures for P2P networks. This represents a first attempt towards a more elaborate and efficient system of parallel event-driven simulation (PEDS) using GPU. In such a PEDS for P2P networks, time is chopped into slots. The events falling into each of these time slots are executed in parallel. Both the advantage of CPU and GPU should be taken to optimize the system performance. CPU could take the responsibility for scheduling the various events in a P2P network, such as peer join/departures, peer selection, data block transmission etc. On the other hand, the advantage of GPU will be exhibited when it executes a large number of parallel events falling in a same time-window. Our ultimate goal is to build a parallel event-driven simulation, leveraging the

strengths of both CPU and GPU, that can scale to meet to the need for simulating real-world P2P protocols.

References

1. A. Legout, G. Urvoy-Keller, and P. Michiardi, “Rarest First and Choke Algorithms Are Enough,” in Proc. of Internet Measurement Conference (IMC) 2006, Rio de Janeiro, Brazil, October 2006.

2. S. Lin, A. Pan, R. Guo, and Z. Zhang, “Simulating large-scale p2p systems with the widstoolkit,” in Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005.

3. G. F. Riley, R. Fujimoto, and M. H. Ammar, “A generic framework for parallelization of network simulations,” in MASCOTS, 1999.

4. J. Cowie and H. Liu, “Towards realistic million-node internet simulations,” in Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, 1999.

5. D. M. Rao and P. A. Wilsey, “Simulation of ultra-large communication networks,” in MASCOTS, 1999.

6. X. Zeng, R. Bagrodia, and M. Gerla, “Glomosim: A library for parallel simulation of large-scale wireless networks,” in Workshop on Parallel and Distributed Simulation, 1998.

7. S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo, “Wids: An integrated toolkit for distributed system development,” in Proceedings of the 10th USENIX Workshop on Hot Topics in Operation System, June 2005.

8. A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, “Scalability and accuracy in a largescale network emulator,” in Proceedings of 5th OSDI, 2002.

9. Ns-2 Simulator Official Website: http://www.isi.edu/nsnam/ns/