Simulating Peer to Peer Networks Using GPU High Perfomance Support

Giovanni Susinni, Giuseppe Greco
University of Catania
Viale A. Doria 6, 95125 Catania, Italy
giovannisusinni@gmail.com, grecoba1984@libero.it

***Abstract*—Peer-to-Peer networks are used by many applications to share resources between nodes. We have proposed a parallel version of a simulator for some aspects of a peer-to-peer network performing file sharing. Being this analysis computationally expensive for contemporary CPUs, the computing power of Graphic Processing Units allows a great gain in performance during simulation. Specifically, we have used the NVIDIA Computer Unified Device Architecture programming model to simulate the behaviour of a peer-to-peer network performing data transfers, and compute communication delays as well as data updates.**

***Index Terms*—P2P networks, GPU computing, CUDA, Performance, Parallel**

I. Introduction

In the last few decades, the Peer-To-Peer (P2P) technology and its many variants, such as BitTorrent, have become popular Internet applications due to their scalability and robustness [1], [2]. Today, the number of devices connected to the Internet is increasing fast. Therefore, it is important to analyze the interactions between peers (or nodes hereafter) and some file sharing issues. The potential huge number of devices in a P2P network may pose a perfomance problem during the simulation of data transfers, and on the other hand simulating the behaviour of novel models for selecting source and destination peers is a current research field.

A sequential simulation of the behaviour of a P2P network is viable for a small amount of data and nodes that transfer data: the results can be achieved in a relatively short time. However, for an extensive analysis, the sequential approach is not ideal for computations, having to cope with a lot of data and transfer nodes. One of the possible solutions is to have a parallel approach using a Graphic Processing Unit (GPU) architecture. The multi-core architecture of a GPU, organized in threads, blocks and grids, shows its great potential for the said simulation thanks to the lower computation time than the sequential approach, particularly when increasing the number of nodes and data transfer (i.e. these are often file fragments). In this paper, we propose to use GPUs to perform a simulation of a P2P network that is closer to the real case. To this end, our work focuses on the implementation, and optimization of exponentiation operations on GPUs.

The contribution of this work is threefold: (i) first, we have developed and analyzed a sequential algorithm, implemented in C language, that simulates some aspects of a P2P network; (ii) second, we have implemented a correspondent GPU algorithm using the capability of a GPU architecture through the Computer Unified Device Architecture (CUDA) C/C++ language; (iii) finally, we have compared the results between the two approaches.

Our paper is organized as follows: Section 2 provides an overview of the P2P model, while Section 3 provides an overview of the CUDA architecture. Section 4 describes the simulation approaches for the sequential and GPU parallel versions. Section 5 shows the experimental results and the comparison of execution measured times. In Section 6 we analyze the related work, and finally Section 7 provides conclusions and possible future work.

II. Background on P2P Networks

An effective network architecture should exhibit the following features [3]:

- Provably good perfomance;
- Low overhead;
- Quality of Service (QoS) of peers;
- Autonomy;
- Robustness.

Over time, the more traditional client-server model based on a centralized approach has become a decentralized architecture for P2P networks [4]. Indeed the continuos growth of users and bandwidth has been accompanied by increasing requests of a diversified wealth of applications, with a remarkable effort in terms of resources to be used to achieve these challenges.

A. Definition and Properties of P2P Networks

A distributed network architecture is a P2P network if its participants share a part of their own hardware resources, as processing power, network link capability, storage capacity [5]. P2P networks are organized as overlay topologies on top of the underlying physical network and are formed by peers connecting to each other in either a structured or unstructured manner [6].

The P2P network must have the following properties [5]:

- *Resource sharing*: each peer contributes with system resources to the operation of the P2P system. Ideally, this resource sharing is proportional to the peers use of the P2P system, but many systems suffer from the free rider problem.

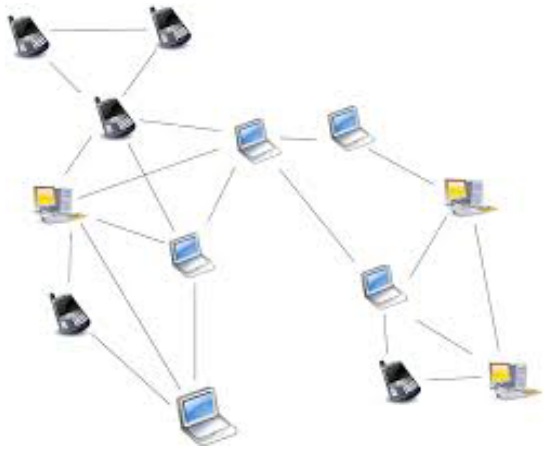

Fig. 1. A P2P network

- *Networked*: all nodes are interconnected with other nodes in the P2P system, and the full set of nodes are members of a connected graph. When the graph is no longer connected, the overlay network becomes partitioned.
- *Decentralization*: the behavior of the P2P system is determined by the collective actions of peer nodes, and there is no central control point. Some systems however secure the P2P system using a central login server. The ability to manage the overlay [7] and monetize its operation may require centralized elements.
- *Symmetry*: nodes assume equal roles in the operation of the P2P system. In many designs this property is relaxed by the use of special peer roles such as super peers or relay peers.
- *Autonomy*: participation of the peer in the P2P system is determined locally, and there is no single administrative context for the P2P system.
- *Scalable*: this is a pre-requisite of operating P2P systems with millions of simultaneous nodes, and means that the resources used at each peer exhibit a growth rate as a function of overlay size that is less than linear. It also means that the response time does not grow more than linearly as a function of overlay size.
- *Self-organization*: the organization of the P2P system increases over time using local knowledge and local operations at each peer, and no peer dominates the system. Biskupski, Dowling, and Sacha [8] argue that existing P2P systems do not exhibit most of the properties of self-organization.
- *Stability*: within a maximum variability rate, the P2P system should be stable, i.e., it should maintain its connected graph and be able to route deterministically within a practical hop-count bounds.

B. System model

Figure 1 provides a conceptual representation of the P2P overlay topology. Since P2P networks are fault-tolerant, not susceptible to single-point-of-failure, P2P overlay topologies are multiply-connected. Broadly, there are three classes of P2P systems:

- *Pure P2P Systems*, in which 2 nodes/devices interact with each other without requiring the intervention of any central server or service.
- *Hybrid P2P Systems*, in which peers rely partially on a central server to provide certain services, although the interaction between peers still takes place independently.
- *Federated P2P Systems*, in which peer interactions take place inside pre-defined domains, such as within an organization.

In this classification we have an increasing degree of centralization.

Moreover, P2P systems can either be: (i) structured [9], where the overlay graph is well structured and a mathematical scheme (e.g. Distributed Hash Tables) is applied to make sure that g nodes are added in a manner which maintains the structure; or (ii) unstructured, where the overlay is a random graph type and new nodes are added to the network in an unpredictable manner. *Structured* P2P Systems are formed on the basis of node-identifiers, guaranteeing information retrieval in bounded time for simple queries and are self-organizing in the face of failures, whereas unstructured P2P systems can support large complex queries, but do not guarantee information retrieval in bounded time with not so efficient self-organization capabilities.

In this paper we consider a *Structured* and *Pure* P2P network.

III. CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables an increase in computing performance by harnessing the power of GPUs. With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. Here are a few examples: (i) identify hidden plaque in arteries (heart attacks are the leading cause of death worldwide); (ii) analyze air traffic flow, the National Airspace System manages the nationwide coordination of air traffic flow. Computer models help identify new ways to alleviate congestion and keep airplane traffic moving efficiently. Using the computational power of GPUs, a team at NASA obtained a large performance gain, reducing analysis time from ten minutes to three seconds. The speed-up is the result of the parallel GPU architecture, which however require developers to port compute-intensive portions of the application to the GPU using the CUDA Toolkit.

CUDA works, conceptually, according to the architectural model shown in Figure 2. The graphic chip, in the CUDA model, is constituted by a series of multiprocessors, called Streaming MultiProcessor. The number of multiprocessors depends on the characteristics specific to the class and performance of each GPU. Each processor can perform a mathematical operation (sum, multiplication, subtraction, etc.) on integers or floating point single-precision numbers (32-bit). In each processor there are also two multi-unit. Special functions

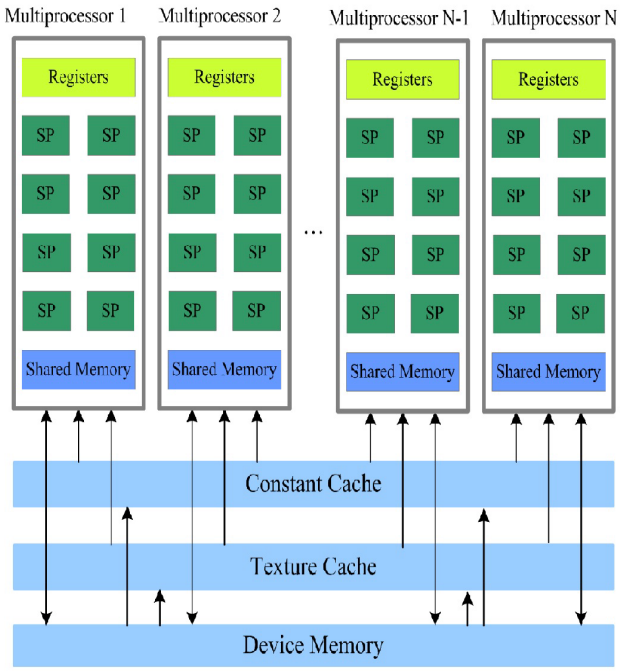

Fig. 2. GPU architecture

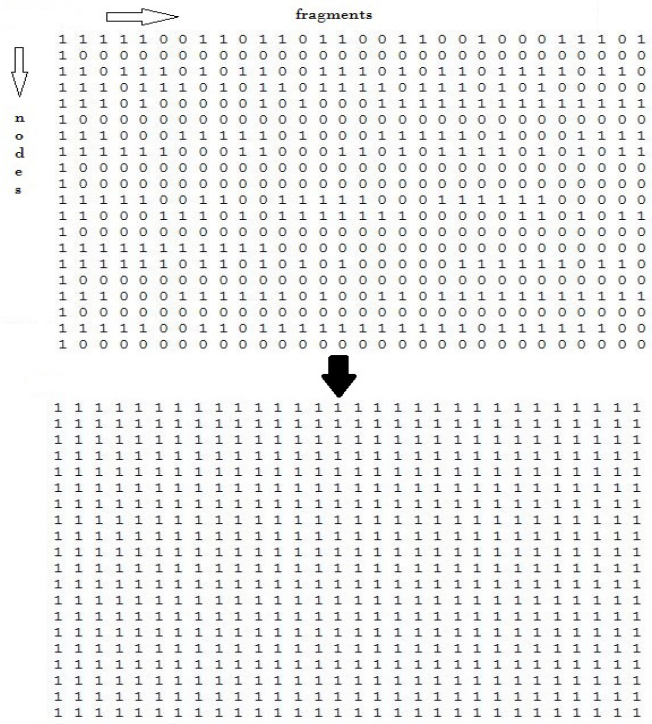

Fig. 3. An example of state update for nodes

are given (that perform transcendent as sine, cosine, reverse etc.) and, only for chips based on GT200 architecture, a single unit in floating-point double-precision (64-bit) is available. In a multiprocessor there is a shared memory accessible by all streaming processors, cache for instructions. The other types of memory, as shown in the picture, are accessible by each processor and represent the main repository for large amounts of data to be saved in the GPU. In the following we analyze the parts of code in the GPU using the libraries provided by NVIDIA.

A. Kernel and hierarchy of threads

CUDA programming indicates as a kernel function a portion of code that runs in parallel, N times.

The individual run of the kernel is performed by a simple unit called thread. CUDA threads are simpler than CPU threads, so the code is considered to be faster. To determine the number of threads that run a single kernel there is a logical organization and a two-level hierarchy. In the top level they depend on the size of a grid. The grid is a two-dimensional level comprising blocks that in turn have a three-dimensional structure that is specified by the number of threads.

Once a kernel is launched it receives two structures specifying the additional parameters for the size of grid and blocks. This listing shows the syntax to launch a kernel function named *kernelfunction()*.

```
dim3 blocksize(x, y, z);
dim3 gridsize (x, y);
kernelfunction <<<gridsize, blocksize>>>(parameters);
```

Listing 1. Kernel syntax

The logical division in grid and blocks is a crucial aspect in the design and code, in CUDA. In addition to the limits imposed by the hardware, there are other precautions that should be taken into account to make code more efficient.

IV. P2P Simulation

A P2P-based application is a distributed application that provides tasks or work loads to peers, which have the same privileges. Peers give a portion of their resources available to other network participants, they need not a central coordination server. A P2P system can be used in many application domains, e.g. in social contexts, like sharing files and resources. Today, given the large diffusion of P2P networks we need to simulate and assess the performance of novel transfer protocols, in terms of the computation time and how many nodes are involved into the communication. Generally, according to the increase of nodes and files, the computation time increases.

A. Issues of P2P Network Simulation

In our simulation, we suppose that each peer communicates with others. Each peer has some file fragments and shares them, while at the same time receives the missing parts of some files. The peers and file fragments are represented in an array with N peers and M fragments, while the latency times of communication between nodes (namely *ping*) are in an array N*M, named `p`.

On top of figure 3 it is shown the initial state of the peers, in terms of availability of file fragments. We can see that a single node has some file fragments: 0 represent the *absent* state, while the 1 represent the *available* state. On the bottom

it is shown the final state of the peers, after the fragments have been transferred between peers.

B. Sequential simulation

The first approach is the sequential one. Listing 2 shows the algorithm in C language, with the selection of peers that will receive the file fragments and update their state (from *absent* to *available*). Constant `N` is the number of peers, and `M` is the number of fragments. Array `rics` contains the destination peer indexes. There are *MAXINS* destination peers for a source peer, which are sequentially counted starting from the index of the source peer.

The update of file fragments consists of comparing for each source peer, having index `tidx`, its fragments availability, as determined by array `f`, with the fragments availability of a peer having index `rics[i]`. If the difference between source and destination is equal to zero, there will not be an update, else the source will send missing fragments to the destination peer. To approximate the real case, in the simulation we use the latency times for each fragment transfer, both in download and in upload. In this analysis we assume that the transfer is ideal, i.e. no loss of packets occurs. The simulation has to consider each source peer, file fragment and destination peer, therefore we have three nested `for` cicles.

This approach is viable for a few peers and fragments, i.e. the results can be obtained in a small amount of time. However, the sequential approach is not appropriate for many peers and fragments, the solution is then a parallel approach using a GPU architecture.

```
void chooseNode(int tidx, int *rics){
  for (int i=0; i < MAXINS; i++)
    rics[i] = ((tidx + i + 1) % N);
}

int checkFr(int *f, int tidx, int *rics, int tidy, int i){
  if (f[tidx*N + tidy] - f[rics[i]*N + tidy]) return 1;
  return 0;
}

void selection(float *p, int *f) {
  int rics[MAXINS];
  unsigned long tidx;
  unsigned long tidy;

  for (tidx=0; tidx <= N; tidx++) {
    chooseNode(tidx, rics);
    for (tidy=0; tidy <= M; tidy++)
      for (int i=0; i < MAXINS; i++)
        if (checkFr(f, tidx, rics, tidy, i))
          if (p[tidx*N + tidy] <= tstep*(tidx*N + tidy))
            f[rics[i]*N + tidy] = 1;
  }
}
```

Listing 2. Sequential algorithm updating fragments availability

C. GPU Simulation

The multi-core architecture of GPU, organized in threads, blocks and grids [10]–[15], shows its great potential by achieving a lower execution time than the sequential approach, particularly with the increase of nodes and file fragments. CUDA is essentially C/C++ language with a few extensions that allow one to execute functions on the GPU using many threads in parallel [10]. In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code that runs on the host can manage memory on both the host and device, and also launches *kernels* which are functions executed on the device. Kernels are executed by many GPU threads in parallel [10], [16]. The use of tokens *BlockIdx.x*, *ThreadIdx.x*, *BlockDim.x* allows us to take advantage of indexes of threads and blocks [14].

Indeed the kernel is executed in parallel by *threads*, which are organized in *blocks*, with a 3D structure, and *grids* with a 2D structure. For simulating a P2P network we use a 2D array, with *ThreadIdx.x* indicating the nodes and *ThreadIdx.y* indicating the fragments. As in the sequential algorithm, we check on each peer whether fragments are available. We use the *device* function *checkFr()* for this, whose body is the same as the corresponding function in listing 2. If the fragment is avalaible, the state of the peer can be updated in due time. This global function runs in parallel in each GPU's core thanks to indexes *tidx* and *tidy*.

```
__global__ void selection(float *p, int *f){
  int rics[MAXINS];
  unsigned long tidx = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned long tidy = blockIdx.y*blockDim.y + threadIdx.y;

  chooseNode(tidx, rics);

  for (int i = 0; i < MAXINS; i++)
    if (checkFr(f, tidx, rics, tidy, i))
      if (p[tidx*N + tidy] <= tstep*(tidx*N + tidy))
        f[rics[i]*N + tidy] = 1;
}
```

Listing 3. Parallel algorithm in CUDA C

The keyword *global* indicates a *kernel* function that runs on the device and is called from host code, and runs in multiple instances on several blocks and threads. Listing 3 shows the kernel function for our simulation.

```
cudaMalloc((void**)&dev_p,N*M*sizeof(float));
cudaMalloc((void**)&dev_f,N*M*sizeof(int));
cudaMalloc((void**)&dev_dt,N*M*sizeof(float));
```

Listing 4. Library function cudaMalloc()

```
cudaMemcpy(dev_p,ping,N*M*sizeof(float),
     cudaMemcpyHostToDevice);

cudaMemcpy(output,dev_f,N*M*sizeof(float),
     cudaMemcpyDeviceToHost);
```

Listing 5. Library function cudaMemcpy()

Library function *cudaMalloc()* allocates a given size of bytes of linear memory on the device and returns `*devPtr`, a pointer to the allocated memory (see Listing 4). The allocated memory is suitably aligned for any kind of variable. Function *cudaMalloc()* returns a *cudaErrorMemoryAllocation* in case of failure [17]. Therefore, we can allocate the necessary space to transfer the data from CPU to GPU. In CUDA C/C++ library function *cudaMemcpy()* allows us to transfer data from *host* to *device* and vice versa [17] (see Listing 6).

In function *main()* we invoke our kernel function *selection()* (see Listing 6), which allows to transfer the data from host to device. This performs the GPU computation, and then there is a *cudaMemcpy()* function call, copying from device to host the data indicating updated nodes.

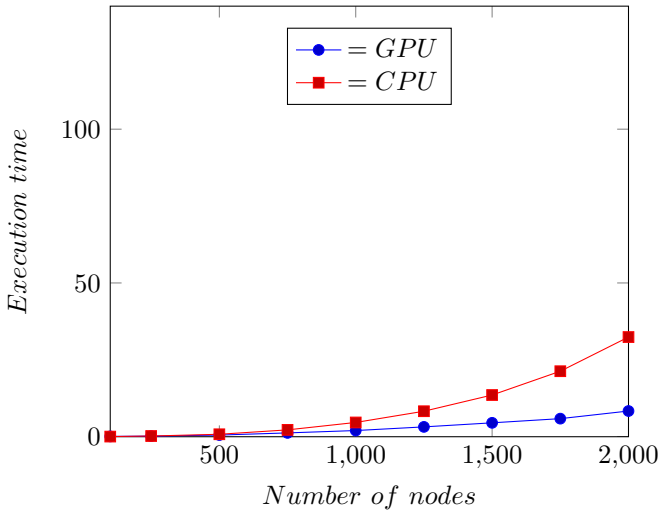

Fig. 4. GPU and CPU execution time with MAXINS=1/25 of total nodes

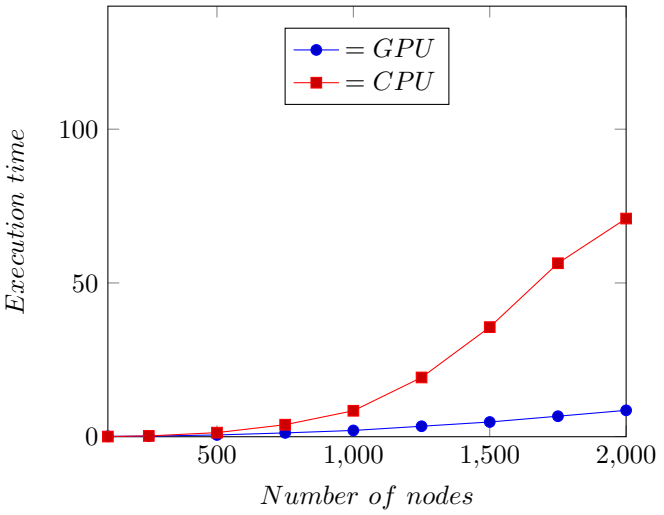

Fig. 5. GPU and CPU execution time with MAXINS=1/10 of total nodes

```
selection <<<blocks, threads>>>(dev_p, dev_f);
```

Listing 6. Calling a kernel function from the CPU program

The input arrays of ping times, node and their fragment availability are initially loaded from files. Every time-step the kernel is called and the output array is updated according to the latency time.

V. Results for the sequential and GPU versions

From the analysis of *sequential* and the *GPU* execution times, we can see that the result is very different with a high number of nodes and file fragments. With a small number of nodes, the sequential computation is faster than the parallel one, because the allocation in memory, that the CUDA compiler does through the GPU's cores, is an expensive process, but this impact is smaller with the increase of nodes and data. The execution time has been obtained from the average of ten measures.

The sequential algorithm for processing data is executed on a AMD(R) Athlon 64 X2 processor with up to 2.9 GHz clock speed and 4GB of DDR3L-1333 RAM, while for the GPU algorithm we used a Nvidia(R) GeForce(TM) GTX 480 with 480 CUDA cores and 1536 MB GDRR5 video RAM [18].

Figure 4, 5 and 6 show the execution times for the sequential (CPU) and parallel (GPU) versions, when considering a number of destination nodes equal to 1/25, 1/10 and 1/5, respectively, of the total nodes N. Indeed with a low N the perfomance of the CPU version are better because for the execution of the GPU version the memory allocation from host to device is a costly operation.

Figure 7 shows the comparison of three simulations, to appreciate the increasing difference in terms of execution time between the two approaches, with a strong dependence on the number of destination hosts *MAXINS*. Table I shows the actual measured times for all considered scenarios.

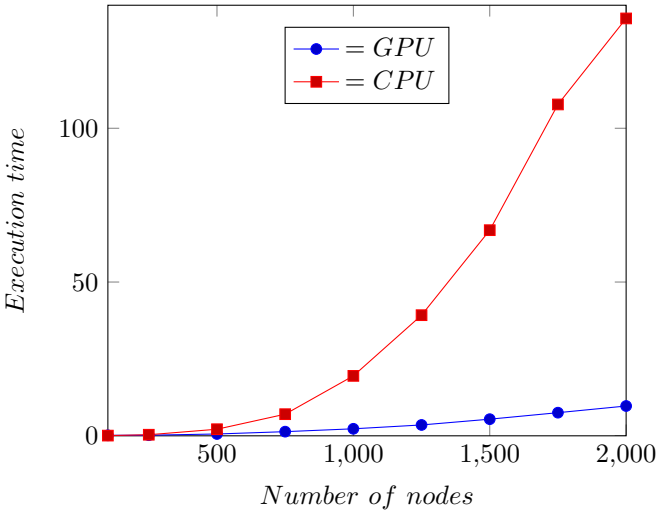

Fig. 6. GPU and CPU execution time with MAXINS=1/5 of total nodes

Figure 8 shows the ratio between execution times on CPU and GPU when varying the numer of *MAXINS*. The highest ratio is when the number of destination peers is highest. This indicates that the parallel version performs much better than the sequential version. Figure 9 provides the execution times on CPU and GPU with 50 destination nodes, when varying the number of peers. GPU performances are alway better than CPU performances, and when the number of peers is the highest also is the gain by resorting to GPU. Table II shows the actual measured times for the latter scenarios considered.

VI. Related Works

Nowadays, computational resources are available on demand and tools are needed to coordinate their use [19]–[21]. Several works aim at proposing ways to make development easier in such complex systems [22]–[27]. Several studies have analysed the behaviour of P2P networks from the point of view

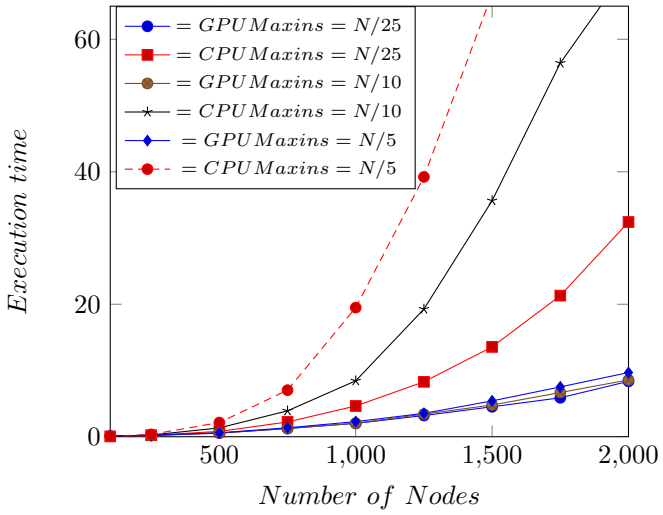

Fig. 7. GPU and CPU execution time: comparison between three cases.

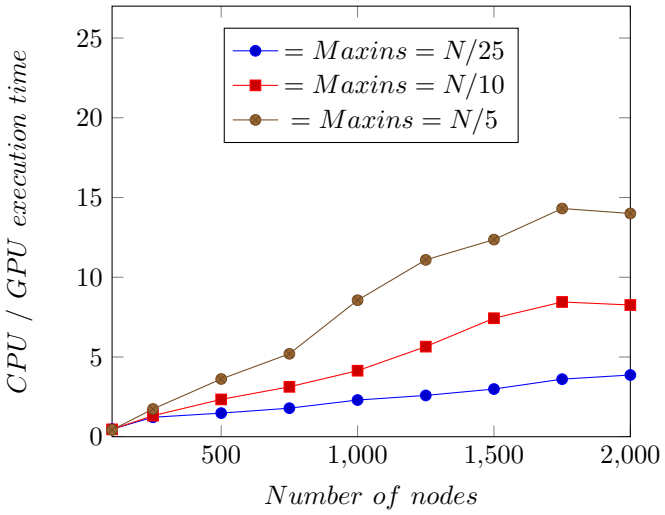

Fig. 8. Ratio CPU and GPU execution times when varying the number of peers for download

TABLE I
EXECUTION TIMES IN GPU AND CPU

Nodes x Fragments	MAXINS	GPU (s)	CPU (s)	CPU/GPU
100x100	4	0.11	0.05	0.48
	10	0.13	0.06	0.46
	20	0.16	0.07	0.43
250x250	10	0.19	0.23	1.22
	25	0.20	0.26	1.32
	50	0.20	0.35	1.74
500x500	20	0.54	0.80	1.48
	50	0.57	1.33	2.34
	100	0.59	2.14	3.62
750x750	30	1.24	2.23	1.79
	75	1.24	3.91	3.13
	150	1.35	7.03	5.2
1000x1000	40	2.01	f4.64	2.3
	100	2.04	8.44	4.14
	200	2.28	19.49	8.56
1250x1250	50	3.20	8.29	2.59
	125	3.41	19.27	5.65
	250	3.53	39.22	11.09
1500x1500	60	4.52	13.56	2.99
	150	4.79	35.64	7.43
	300	5.41	66.86	12.36
1750x1750	70	5.88	21.30	3.61
	175	6.67	56.41	8.45
	350	7.52	107.74	14.32
2000x2000	80	8.36	32.41	3.87
	200	8.58	70.94	8.26
	400	9.69	135.77	14.00

TABLE II
EXECUTION TIMES IN GPU AND CPU WITH MAXINS=50

Nodes x Fragments	GPU (s)	CPU (s)
100x100	0.10	0.10
250x250	0.21	0.35
500x500	0.7	1.29
750x750	1.15	2.96
1000x1000	2.14	5.25
1250x1250	3.12	8.36
1500x1500	4.44	11.82
1750x1750	6.29	16.51
2000x2000	7.88	21.51

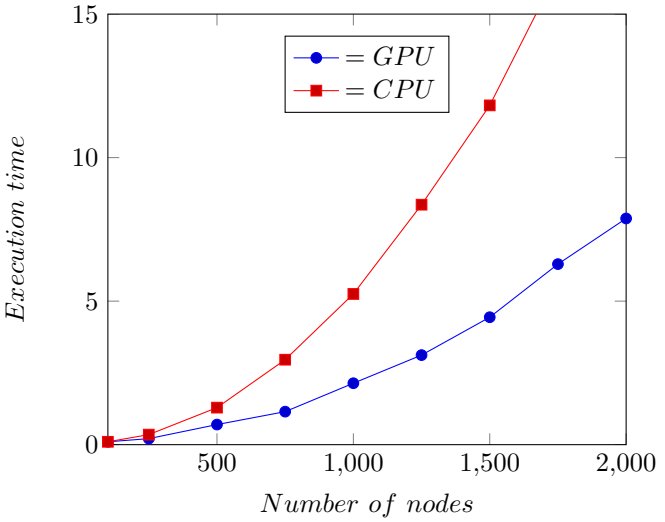

Fig. 9. GPU and CPU execution times with MAXINS=50 nodes

of shared files, i.e. how to have users contribute with contents that can be uploaded by other users, respecting the latency time in download and upload.

Some works have studied the problem of computation in P2P networks using the GPU approach vs the CPU approach. In [6] authors analyze that heterogeneous nodes can have multiple types of computing elements, and the performance and characteristics of each computing element can be very different. For the simulation of the shared file fragments, we assume an ideal case without giving priority to any file fragments. Instead in [2], [28], authors propose to apply a mathematical model for the diffusion of fragments on a P2P in order to take into account both the effects of peer distances and the changing availability of peers over time.

In this paper we developed two approaches: the first one is the sequential one, which we have implemented as C

language. The second is the parallel approach, which uses CUDA C/C++ language. In [29] some authors present FLAP, a tool to generate CUDA parallel code from sequential C code. This tool uses patterns to generate parallel CUDA code.

In our paper an important focus is the performance in terms of time, with a comparison between GPU and CPU computation. In [13] the authors proposed three parallel algorithms that maximize the parallelism of processes, i.e., the power of GPUs is fully utilized. With their implementation of the algorithms, they are able to achieve up to 12 times the speedup over the highly optimized CPU counterpart, using the NVIDIA GPU and the CUDA programming model.

VII. Conclusion

We have presented a sequential and a parallel solution for the simulation of a P2P network and tested them in an ideal scenario, i.e. without collisions or missing data, and achieved considerable perfomance gains by using parallelism. Indeed in each simulated setting we appreciated that the GPU solution has a lower execution time than the sequential one for updating a peer state. This becomes more evident with the increase of peers and fragments. Particularly, the perfomance of the sequential approach is worst when increasing the number of destination peers, having an exponential increase of the execution time, versus the linear trend of a GPU solution.

In the future we can improve the simulation by including non-ideal effects as collisions, higher latency times in uploads and downloads, and the transfer of larger data other than our binary data.

References

[1] K. Zhao, X. Chu, M. Wang, and Y. Jiang, "Speeding Up Homomorpic Hashing Using GPUs Communications," in *Proceedings of IEEE International Conference on Communications (ICC)*, 2009, pp. 1–5.

[2] C. Napoli, G. Pappalardo, and E. Tramontana, "Improving files availability for bittorrent using a diffusion model," in *Proceedings of IEEE International WETICE Conference*, Parma, Italy, June 2014, pp. 191–196.

[3] Y.-K. R. Kwok, *Peer-to-Peer Computing: Applications, Architecture, Protocols, and Challenges*. CRC Press, 2011.

[4] R. Steinmetz and K. Wehrle, *Peer-to-Peer Systems and Applications*, ser. Information Systems and Applications. Springer, 2005, vol. 2485.

[5] X. Shen, H. Yu, J. Buford, and M. Akan, *Handbook of Peer-to-Peer Networking*. Springer, 2010.

[6] A. Gupta and L. K. Awasthi, "Peer-To-Peer Networks and Computation: Current Trends and Future Perspectives," *Computing and Informatics*, vol. 30, no. 3, pp. 559 – 594, 2012.

[7] J. Buford, "Management of peer-to-peer overlays," *International Journal of Internet Protocol Technology*, vol. 3, no. 1, pp. 2–12, 2008.

[8] B. Biskupski, J. Dowling, and J. Sacha, "Properties and mechanisms of self-organizing MANET and P2P systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 2, no. 1, p. 1, 2007.

[9] M. Bawa, B. F. Cooper, A. Crespo, N. Daswani, P. Ganesan, H. Garcia-Molina, S. Kamvar, S. Marti, M. Schlosser, Q. Sun *et al.*, "Peer-to-peer research at stanford," *ACM SIGMOD Record*, vol. 32, no. 3, pp. 23–28, 2003.

[10] *NVIDIA CUDA*. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[11] *Compute Unified Device Architecture: Programming Guide*, 2nd ed., jun 2008.

[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," in *Proceedings of the IEEE*, vol. 96, no. 5, 2008, pp. 879–899.

[13] X. Chu, K. Zhao, and M. Wang, "Massively Parallel Network Coding on GPUs," in *Proceedings of IEEE International Performance, Computing and Communications Conference (IPCCC)*, 2008, pp. 144–151.

[14] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[15] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*. Morgan Kaufmann, 2012.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[17] *NVIDIA CUDA Library Documentation*. [Online]. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/index.html

[18] *NVIDIA GeForce GTX 480*. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications

[19] G. Borowik, M. Woźniak, A. Fornaia, R. Giunta, C. Napoli, G. Pappalardo, and E. Tramontana, "A software architecture assisting workflow executions on cloud resources," *International Journal of Electronics and Telecommunications*, vol. 61, no. 1, pp. 17–23, 2015.

[20] C. Napoli, G. Pappalardo, and E. Tramontana, "An agent-driven semantical identifier using radial basis neural networks and reinforcement learning," in *XV Workshop "From Objects to Agents" (WOA)*, vol. 1260. Catania, Italy: CEUR-WS, September 2014.

[21] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappalà, "A cloud-distributed gpu architecture for pattern identification in segmented detectors big-data surveys," *Computer Journal*, 2014. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxu147

[22] R. Giunta, G. Pappalardo, and E. Tramontana, "Using Aspects and Annotations to Separate Application Code from Design Patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Sierre, Switzerland, March 2010.

[23] ——, "Aspects and annotations for controlling the roles application classes play for design patterns," in *Proceedings of IEEE Asia Pacific Software Engineering Conference (APSEC)*, Ho Chi Minh, Vietnam, December 2011, pp. 306–314.

[24] A. Calvagna and E. Tramontana, "Delivering dependable reusable components by expressing and enforcing design decisions," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 493–498.

[25] R. Giunta, G. Pappalardo, and E. Tramontana, "Superimposing roles for design patterns into application classes by means of aspects," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Riva del Garda, Italy, March 2012, pp. 1866–1868.

[26] E. Tramontana, "Automatically characterising components with concerns and reducing tangling," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 499–504.

[27] A. Fornaia, C. Napoli, G. Pappalardo, and E. Tramontana, "An AO system for OO-GPU programming," in *XVI Workshop "From Object to Agents" (WOA)*, vol. 1382. Napoli, Italy: CEUR-WS, June 2015, pp. 24–31.

[28] C. Napoli, G. Pappalardo, and E. Tramontana, "A mathematical model for file fragment diffusion and a neural predictor to manage priority queues over bittorrent," *International Journal of Applied Mathematics and Computer Science*, vol. 26, no. 1, 2016.

[29] E. Hernandez Rubio, A. Meneses Viveros, P. M. C. Perez, S. D. H. Zavala, and H. M. M. Rios, "Flap: Tool to generate cuda code from sequential c code," in *Proceedings of International Conference on Electronics, Communications and Computers (CONIELECOMP)*. IEEE, 2014, pp. 35–40.