Hybrid CPU-GPU Distributed Framework for Large Scale Mobile Networks Simulation

Ben Romdhanne Bilel; Nikaein Navid; Mohamed Said Mosli Bouksiaa
Eurecom 2229, route des Cretes
06904 Sophia Antipolis, France
benromdh,nikaeinn,mosli@eurecom.fr

***Abstract*—Most of the existing packet-level simulation tools are designed to perform experiments modeling a small to medium scale networks. The main reason of this limitation is the amount of available computation power and memory in quasi mono-process simulation environment. To enable efficient packet-level simulation for large scale scenario, we introduce a new CPU-GPU co-simulation framework where synchronization and experiment design are performed on CPU and node's processes are executed in parallel on GPU according to the master/worker model [13]. The framework is developed using Compute-Unified Device Architecture (CUDA) and denoted as Cunetsim [18], CUDA network simulator. To study the performance gain when GPU is used, we also introduce the CPU-legacy version of Cunetsim optimized for multi-core architecture.**

In this work, we present Cunetsim architecture, design concept, and features. We evaluate the performance of Cunetsim (both versions) compared to Sinalgo and NS-3 using benchmark scenarios [20]. Evaluation results show that Cunetsim execution time remains stable and that it achieves significantly lower computation time than CPU-based simulators for both static and mobile networks with no degradation in the accuracy of the results. We also study the impact of the hardware configuration on the performance gain and the simulation correctness.

Cunetsim presents a proof of concept, demonstrating the feasibility of a fully GPU-based simulation rather than GPU-offloading or partial acceleration, through adequate architecture.

***Keywords*-Large Scale; Hybrid simulation; GPGPU; CUDA; Parallel simulation;**

I. Introduction

Packet-level simulators are usually based on a discrete event paradigm where sequences of events are generated. In general, such events represent mobility, connectivity, channel calculation and in/out packets processing. The time complexity and memory usage of a simulation are then proportional to the frequency of these events for the total number of nodes, which represent the main bottlenecks when targeting scalability and efficiency. There exists also a trade-off between the accuracy of the models, in particular channel models, and time complexity that has to be taken into account when targeting large scale simulations. This calls for a parallel node execution environment with minimal inter-processes communication overhead [9].

In the literature, there are three major approaches to deal with large scale simulation: (i) CPU-based parallel & distributed simulation, (ii) Partial acceleration using specific Co-processor and (iii) The fully GPU approach.

In a CPU-based parallel and distributed simulation [14], the platform may be federated and includes multiple copies of the same or different simulators (modeling different portions of the network) linked together either sequentially or in parallel. Such a federated approach makes use of the existing models and provides a rapid parallelization of existing sequential simulators [17]. However, such approach introduces a significant overhead due to the synchronization among different processes and/or machines and requires sophisticated and expensive simulation infrastructure [13]. This overhead may increase drastically in mobile environment if the network topology and machines mapping is not dynamically managed (e.g. through nodes migration). For the majority of CPU-based simulators, the performance degradation happens when a combination of the limiting factors, mobility rate, number of nodes, and traffic load increases. For the distributed simulators, such performance degradation happens when the inter-machines communication increases. A scalability demonstration, based on the distributed NS-3 has carefully avoided the problem of the interaction between nodes in different simulation machines [11]. Even if parallel and distributed simulators have crossed a scalability boundary, they introduce new problems such as the cost of a simulated node, the strategy of initial nodes distribution and their migration across different machines.

The second approach addresses the question differently, It aims to increase the efficiency of the simulation locally by offloading the most CPU-intensive part of the simulation from the CPU to a dedicated co-processor. The FPGA was widely used as an acceleration solution [8] however, in some recent approaches, the Graphics Processing Unit (GPU) is used to offload intensive computing tasks such as channel modeling [5] and queuing [15] within the simulator. Recent studies of GPUs allow us to utilize the GPU for more general-purpose computation (GPGPU) [16], or even as a GPU-accelerated simulation architecture when accuracy and runtime performance are both critical [4]. Thus, the GPU has become an increasingly attractive alternative to the expensive CPU-based parallelism, with significant computational power at a relatively low cost. With the advent of the GeForce 8 series GPU in 2006 and the compute unified device architecture (CUDA) [12], the control of the unified stream processors of GPU is transparent to the programmer, and CUDA provides an efficient and wealthy environment to develop parallel codes in a high-level language without the need for graphics-specific knowledge. Even if this approach reduces significantly the computing time, the simulation remains principally in the CPU which continues to be the main system bottleneck in large scale scenarios. Further, a continuous transfer between the GPU memory and the CPU one presents a serious limitation of such approach.

The third approach aims to realize the simulation entirely on the GPU which reduce significantly the memory transfer compared to the second approach and decrease the synchronization latency compared to classic parallel approach. However, the GPU is not fully X86 compliant and did not support CPU features, needs a specific software architecture to disclose its power and did not support memory lock mechanism. Because of these limitations, the fully GPU simulation approach is poorly studied even if it is extremely promising in term of raw performance. As a proof of concept, we propose to use the GPU as a main simulation environment and the CPU as a controller, introduced VIA a new CPU-GPU co-simulation framework denoted as Cunetsim, CUDA Network Simulator. Cunetsim is an experimental simulation platform allowing validation and experimentation of a novel approach. As opposed to previous works, Cunetsim is designed to provide an independent parallel execution environment for each simulated node. Nodes communicate with each other only through the message passing based on the buffer exchange, thus avoiding the usage of any global knowledge on one hand, and increase significantly the parallelism level on the the other hand. Furthermore, it exploits the master/worker model for CPU-GPU co-simulation and provides hybrid synchronization model which maximizes the efficiency and respects the correctness of the simulation. The simulation exploits the large number of computing cores of the GPU to execute nodes in parallel and the high speed memory access to reduce nodes communication latency.

The remainder of the paper is organized as follows. Section II presents the framework architecture and features. Preliminary comparative results are given in section III. Detailed study of the hardware configuration impact is summarized in sectionIV and we discuss limits of our concept in section V followed by concluding remarks and future directions in section VI.

II. The Cunetsim Framework

Cunetsim framework is designed and implemented following a hardware/ software co-design approach to maximize the efficiency. The simulation distribution is based on the master/ worker model [13] where the master controls the simulation achieved by the workers group. Figure. 1 summarizes the cunetsim components' hierarchy through three blocks: the master, the worker and common APIs.

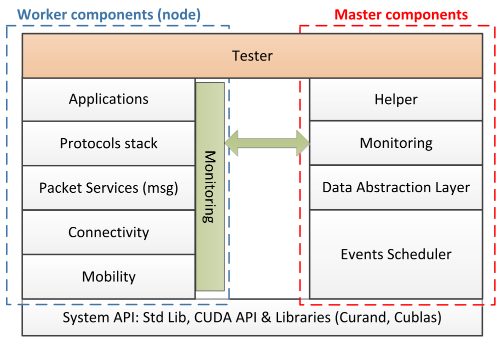


Figure 1. Cunetsim Framework Architecture and Dependency

Conceptually, a worker is associated with one node and is therefore composed of a node's five Worker Processes (WPs) [6]: (i) the application, (ii) the protocol stack, (iii) the mobility, (iv) the connectivity and (v) the packet services.Their functioning is explained in section II-A. The master is also composed of five components: (i) The hybrid events scheduler, (ii) The data abstraction layer, (iii) The scenario manager, (iv) The monitoring component and (v) The helper . The detailed implementation is explained in section II-B. Common APIs regroup those shared by the master and workers. It includes three components: (i) system API, including CUDA APIs and libraries, (ii) monitoring API, and (iii) testing API.

A. The Worker Design

The Worker implements the simulated node, modeled as a stack of independent WPs. Nodes communicate through messages passing. Only buffers are exchanged between nodes to avoid global knowledge. In Cunetsim, each node contains five ordered WPs described in the following sections.

1) Applications (APP): Cunetsim provides a packet-level traffic generator to simulate application data based on packet size and inter-departure time. Each instance is completely independent, allowing the framework to support an important load. The traffic generator tags the packet as a function of communication type: unicast, multicast and broadcast, and assumes that such traffic will be processed by the PROTO and PKT WPs.

2) Protocol stack(PROTO): implements the node behavior both in control and data planes, which are protocol or algorithm specific. It may also include additional models required by such a protocol. Cunetsim implements various broadcasting techniques, such as probabilistic, counter-based and location-based. Such implementations support GPU parallelism and provide inter-process communication through a buffer exchange, avoiding simulation global knowledge, to ensure the simulation scalability and efficiency.

3) Mobility (MOB): The MOB calculates a specific movement in the defined space following a mobility model, for each node. We define a generic mobility container, implemented as a unique CUDA kernel which functioning is explained in Figure.2 . We implement two mobility models: *RandomWayPoint* and *RandomDirection* [10] and three boundary policy models: *Annulment of excess*, *Sliding on the boundaries* and *bouncing on the boundaries*.

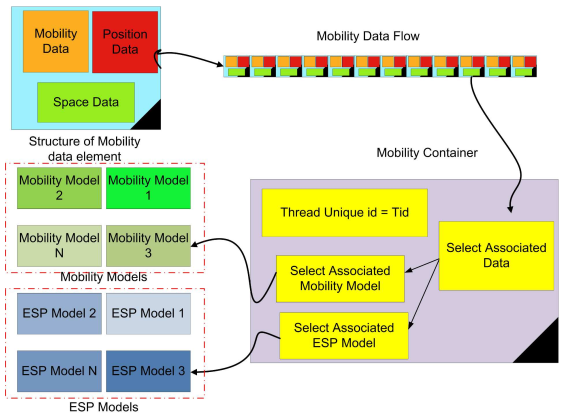


Figure 2. MOB WP functioning

4) Connectivity (CON): The connectivity WP identifies all neighbors of the concerned node. This problem is NPC [7]. The complexity of the brute force approach is of the order of $O(N^2)$. In Cunetsim design, we divided the space into geometric cells where the radius of the cell must be at least the double of the maximum transmission range ($2 * Rmax$). In this case each node will find its neighbors in its own cell and in the neighboring cells. This approach reduces significantly the complexity, which will be related to the network density. We define a connectivity container, which will call a specific connectivity model (the Unit Disk Graph *(UDG)* and the Quasi Unit Disk Graph *(QUDG)* are available in the first version). This kernel will be instantiated into N GPU threads, where N is the nodes number in the scenario. The pseudo-code of the connectivity WP is summarized in the algorithm 1. Each node will be identified with its tid and will be executed independently. By calling the ParseCell function, we apply an optional optimization. Cell is the data structure used to represent the geometric cell and its member which name is ”member” contains the ids of the nodes that currently belong to the cell. Using these variables, a node is able to access to each node contained in a particular cell in order to check their mutual distance.

5) Packets services (PKT): Packets services manage their exchange between nodes. The notion of packet can represent any protocol data unit (PDU), which is layer-dependent. To simulate multiple interfaces, a node may have more than

```
tid=BlockDim.x*blockIdx.x+threadIdx.x;
MyCell=Node[tid].Cellid;
NeighborCell=Cell[MyCell].Neighbor;
for i of NeighborCell do
    if ParseCell(Node[tid],Cell[i]) then
        Continue;
    end
    Nnodes=Cell[i].size;
    for j of Nnodes do
        candidate=Cell[i].member[j];
        if UDG(Node[tid],Node[Candidate]) then
            Node[tid].neighbor[Node[tid].V]=candidate;
            Node[tid].V++;
            if(Node[tid].V==MaxNeighbor)Break;
        end
    end
end
```

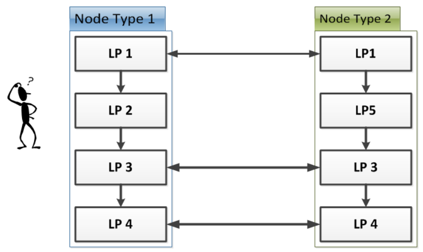
Algorithm 1: The Pseudo-code of the Connectivity WP

one buffer, each of which associated with a given interface. Packet services support both send and receive. Packet send service allows a node to write a packet to the selected in-buffer of the neighbor(s). The packet write operation is an atomic operation avoiding the destination in-buffer to be over-written, as described in section II-B2. It has to be mentioned that the sending process adds a simulation header with additional relevant information including the timestamps, the sending energy, and antenna characteristics (e.g. orientation, type), which is used at the receiver. Packet receive service allows a node to read at most one packet from its in-buffer at each simulation time (i.e. round). However, the in-buffer is capable of receiving up to M packets from other nodes at each round. The receiving service determines which message has to be read by the node based on the lowest timestamps and/or signal energy derived from the simulation header.

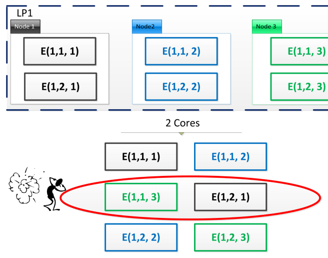
B. The Master Design

The master ensures the simulation correctness, simplifies the framework usability by providing high-level simulation APIs, and guarantees the simulation reproducibility. These features are performed via five components detailed below:

1) Hybrid Events Scheduler: Cunetsim events scheduler (CES) implements a conservative approach for all dependent WPs where we respect a strict order between sequential WPs for each node. This model was developed in [19] where the notion of *WP Pool* is introduced: a WP pool, Π_i is defined incorporating same WPj for all nodes. For a given Π_i, all P_{ij} processes must end to assert that Π is achieved. This presents a simple yet efficient implementation of the coherence and consistency paradigm.In addition, CES addresses two fundamental scheduling issues: independent WPs on one hand and the events sequencing of each WP on



(a) WPs dependency: while the scheduler can determine that WP3 threads cannot start until finishing those of WP2 and WP5, there is no strict order between them.



(b) Events Conflict: E113 and E121 will be executed at the same time while the strict synchronization prohibited

Figure 3. Scheduling Ambiguities

the other hand.

(1) Independent WPs concerns typically heterogeneous simulation and happens when nodes are composed of different WPs sequences. Figure. 3(a) shows a situation where we have two kinds of nodes which implement two independent WPs: WP2 and WP5. In such case, conservative approach did not define a deterministic order between WP2 and WP5 pools. As Cunetsim targets to maximize the efficiency, in such situation we use an optimistic approach, where both of WP2 and WP5 pools can be executed in parallel on the GPU.

(2) The sequencing of WP's events typically happens when the number of cores is not a multiple of the number of nodes. Figure. 3(b) shows one WP composed of 2 events for 3 nodes and 2 cores. In this figure each event is identified by a triple (WP_id, Event_id, Node_id). In conservative and optimistic approaches, the execution of the event (1,2,1) at the same time as (1,1,3) is forbidden since a strict order exists. However respecting this order may induce a significant waste of resources (e.g. 25%in this example), while executing both of them at the same time will not impact the correctness of the simulation. In such case, Cunetsim applies a relaxed approach within the events of each WP's pool. In this way, the WPs order and the events sequencing of each node will be preserved, thus maximizing total resource usage.

To summarize, the Cunetsim hybrid event scheduler works as following: The conservative approach is used for sequential WPs as defined by the simulation model. The optimistic approach is applied when possible, especially for independent WPs and independent WPs sequences. Relaxed approach is applied for event scheduling into each WP.

It has to be mentioned that the CES benefits from the GPU hardware scheduling capabilities. Indeed, the optimistic approach is achieved using the GigaThreads scheduler (i.e. GPU hardware acceleration), the relaxed approach using the 4 wrap schedulers of each SM. The conservative approach is implemented in software.

2) Data Abstraction Layer: Cunetsim data are modeled based on the kernel/flow model. We define several flows where each one presents a specific part of the simulation data. Data is grouped by functionality. Each WP uses one (or more) flow and each node has a specific box with R/W rights. One node can access foreign data with read right. Flow model is natively used by graphics application to manage the communication between the GPU and the CPU. We apply a flows loading-offloading mechanism between the GPU memory (limited and non-extensible) and the principal memory (larger and extensible). The master manages flows transfer between the principal memory and the GPU one, such that no WPs will be in famine situation.

The memory management component provides two services: memory allocation abstraction (MAA) and critical section management (CSM). MAA insures the double allocation of each data flow in both of the RAM and the DRAM. The synchronization of the two copies of the flow is a manual operation which must be specified by the user. Critical section is a recurrent challenge in case of shared memory between several processes. Software mutual exclusion solutions such as semaphores, mutex and locks are commonly used in CPU context. However, GPU context did not provide such explicit solutions. The problem arises mainly when two nodes try at the same time to write messages in a third node's in_buffer, in which case we may lose some of them. CSM provides an abstraction of this problem based on CUDA atomic operations: thanks to atomicInc, a node makes an atomic reservation operation before proceeding to the writing of the message. This operation consists in atomically incrementing the write_index, a pointer to a box in the receiver's buffer.

3) Scenario Management: This component ensures the reproducibility of the simulation via a complete XML layout incorporating five sub-categories: the system, the environment, the network topology, the application configuration and the simulation I/O. This concept aims to simplify the interaction between the simulation and the user. The process applied for cunetsim reflects the same experimentation workflow proposed on [6].

C. Common APIs

Common APIs are those shared by the master and the worker and include system/host, Cuda, monitoring and testing tool APIs and libraries. Cuda API includes the driver and the runtime used to manage the GPU using high level programming language such as C/C++. Cuda libraries provide an efficient hardware acceleration of common libraries such as Math and BLAS libs. The monitoring process is a CPU-expensive task which may reduce the efficiency of the simulation and introduce an important overhead. Since Cunetsim benefits from at least two computing contexts, it could easily offload this process from one simulation context to another depending on the load (e.g. from CPU to GPU or vice versa, or even from one GPU to another). Cunetsim monitoring provides three APIs as follows:

GPU Monitoring: Each WP uses, in addition to simulation data flows, monitoring data flows, where WP instances write their monitoring results (e.g. number of messages, processing time, flags). A specific monitoring WP is implemented and used to process these flows to produce final results. In this approach, the monitoring process is included in the simulation and thus necessarily impacts its performances, however, the impact can be reduced using dedicated device (second GPU). Such approach is appropriate to online monitoring techniques since it provides results as soon as they exist.

CPU Monitoring: The monitoring process dumps -in asynchronous mode- the simulation data flows into the RAM to process them and produces final monitoring results. The asynchronous dumping operation will not impact the simulation performance, however, the CPU process must be able to consume these flows in the same speed (or higher) that the simulation produces; otherwise the RAM will be saturated. This approach is more appropriate to offline monitoring technique since it can use saved data.

Co-GPU-CPU Monitoring: In this approach, the monitoring process is shared between GPU and CPU. As in the GPU approach, WPs use monitoring data flows to write raw information, and as in the CPU approach, the CPU process dumps in asynchronous mode the monitoring flows. This approach impacts moderately the simulation performance but reduces significantly the size of transferred data, making this approach more adequate for a large scale scenario for both offline and online monitoring.

1) The Tester API: The tester API implements a validation component which ensures the simulation correctness, in particular the user-specific implementation for a given WP and its integration with the simulation framework. The basic test consists of implementing the same algorithms, sequentially and in parallel, for both master and worker. The master tests the process, compares both of their results and validates the worker group results. It has to be noted that the testing process is only used in developing and debugging mode when the simulation correctness is required.

D. Hardware Mapping

In this section, we detailed the hardware mapping of the cunetsim's software components: First we present the hardware mapping of the fully GPU version and second we describe the CPU-legacy one.

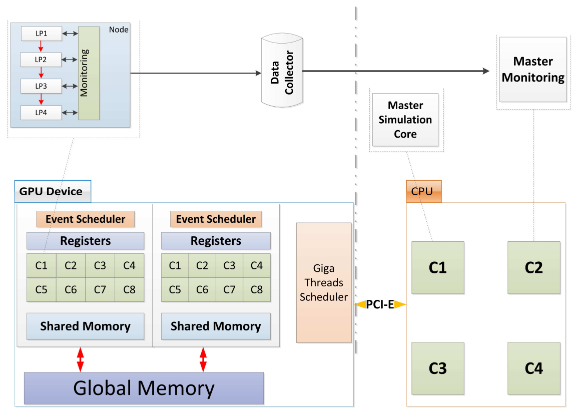


Figure 4. Cunetsim Hardware Mapping

1) Fully GPU version: The hardware mapping of the software components is presented in Figure.4, further detailing the CPU-GPU co-simulation. At each time, it can be seen that each WP is mapped to a GPU core, called scalar processor (SP) wrapped within a streaming multiprocessor (SM), while the master processes are mapped to CPU cores. Nodes' WPs exploit three memory levels: registers, shared and global memory. Registers include local variables of each WP instance. Shared memory is used as an acceleration cache where a prior knowledge on the data is available. Global memory is used when WPs communicate (sending messages / reading position) without having a prior knowledge on communicating nodes. It has to be mentioned that such mapping provides a dedicated execution environment for each node, where inter-WP and node communication is minimized over three memory stages. As for the Master, it is represented by two processes: (i) the simulation core and the Data abstraction layer representing the primary process of the master, and (ii) monitoring process in charge of data collection and user interface representing the secondary process of the master. This separation maintains a strict priority order between the primary and the secondary process.

2) CPU-legacy version: Cunetsim architecture is designed for a fully GPU simulation as detailed in section II-D1. However the GPGPU is a recent discipline and rare are the data-centers which use the GPUs as computing Co-Processors. On the other hand, Current and future CPUs are also multi-core and provide interesting features, as the vector parallelism. These reasons have convinced us to provide a pure CPU solution. Based on the PGI unified Binary technology [3], we generate a CPU compliant version which parallelizes nodes through the OpenMP API using several threads. The user can specify the number of threads in conformity with the CPU capabilities. We note that the software architecture and code did not change, only the compilation procedure is different. in following, this version will be appointed as Cunetsim-LN, where the last number presents the threads' number.

III. Comparative Performances Results

To evaluate the real performance of each approach presented in section I under large scale condition regardless of different models impact, we extend the benchmarking methodology for network simulators presented on [20] to support wireless and mobility conditions. In this methodology, authors implement identical node model for all considered simulators. They demonstrate that NS-3 [2], Jist and Omnet++ have the best performance. However, they did not address mobility issues and ignored Sinalgo [1], known as a stable simulator on large scale conditions. In the following study, we choose Sinalgo as a representative CPU-based solution while NS-3 is involved as the most optimized public simulator, providing also a stable distributed version over MPI. The CPU version of cunetsim, which involves 4 CPU cores is a representative case of the partial accelerated approach while the GPU version presents the fully GPU approach. The mobility and the connectivity algorithms are the same as we propose for all simulators. Only a simple flooding protocol is implemented using equivalent algorithms. We propose two benchmark scenarios: The first compares the performance of each simulator's kernel regardless of the efficiency of implemented models, while the second addresses their robustness in mobile conditions. The first scenario models a simple network, where the nodes are arranged in a grid topology as illustrated in Figure. 5. It includes one traffic source which generates 600 uniform

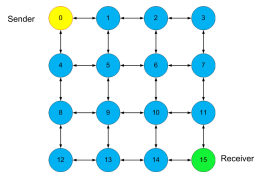


Figure 5. Simple Grid Topology

packets with 1 second of inter-departure time. Packet size is fixed to 128 Bytes. All nodes -including the source- relay unseen packets after a delay of 1 second, thus flooding the totality of the network. The delay of 1 second models the propagation. Nodes do not provide any packets management services. Transmission and reliability are modeled on the channel using a fixed dropping probability which is identical on all links. The sender is the node with the lowest identity and the receiver is the one with the highest identity. In the second scenario, nodes are mobile. The mobility model is the random way point with speed uniformly distributed between $1-5m/s$. The maximum transmission range, Rmax, is 100 and the connectivity model is UDG. The simulation space is a cubic free space whose dimensions are $1600*1600*200$ m. Each node moves before each round and recalculates its connectivity set. Both of these scenarios are outlying real networks and include major node simplifications nevertheless, they have two advantages: they guaranteed a relevant and neutral comparative since they minimize the models' efficiency impact and they provide a representative estimation of the computing power needed for such simulations.

All simulation runs were conducted using a simple PC including an INTEL i7 940 CPU (4 cores with hyper-threading), 6GB of DDR3 and one GPU: the GeForce 460 1GB (336 cores for GPGPU computing). 4 machines are used for the distributed NS-3, interconnected VIA a Gigabyte switch. The OS is Ubuntu Linux 11.10, the Java version is 1.6 and the Nvidia driver version is 285.05.33. Our measurements were taken using NS-3.13, Sinalgo 10.75.03 and Cunetsim prototype. To validate the model equality, we

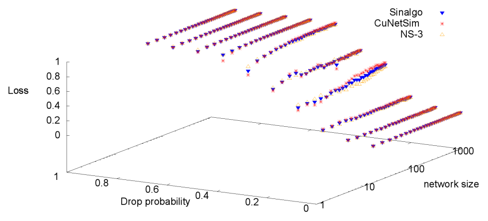


Figure 6. End-to-End Packet Loss: Under the same conditions, all simulators present equivalent E-TO-E loss, considered as the output

use the first scenario with all simulators where we varied the drop probabilities in the interval $[0,1]$. Figure.6 depicts the end-to-end packet loss repossessed and normalized from different simulators, given the dropping probability and the network size. All studied simulators produce similar results.We conclude that our implementations are equivalent -in terms of output- to those of [20]. We evaluate simulators' efficiency regarding two performance metrics: simulation runtime and memory usage. Our results give the average of five executions. The minimal simulation time is set to 700 seconds.

A. Simulation runtime

To evaluate the simulation runtime of the concerned simulators, we fixed the drop probability to 0.1 and we increased the network size from 4 to $102K$ nodes. Section III-A1 analyzes the first scenario results while the section III-A2 addresses the second one.

1) Static Scenario: Figure. 7 shows the average simulation runtime for each simulator. For small to medium networks, Cunetsim-L4 is the fastest simulator up to 2000 nodes and remains faster than Sinalgo and NS-3 in all cases. Beyond, Cunetsim (GPU) becomes the most efficient simulator and the deviation is growing with the network size. Function of the simulators runtime, we distinguish

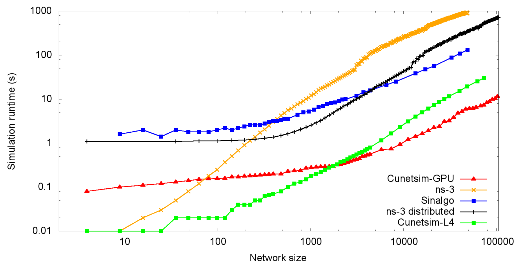


Figure 7. Simulation runtime of the static network: CPU simulators are efficient on small scale but computing power becomes critical on large scale

four network size intervals: small networks [2-50], medium networks [50-200], large networks [200-2000] and the very large networks [2000-102000].

For small scale Cunetsim-L4 and NS-3 are the most efficient simulators. In fact, much of NS-3 components use the very high-speed L3 cache compared with much slower RAM while the cunetsim-L4 uses also the L3 cache and all CPU cores. Cunetsim-GPU is outperformed for two reasons: first the impact of data transfer between the RAM and the GDRAM is significant, second the GPU is underused since only few cores are active. For medium scale, both versions of cunetsim are faster than NS-3. In fact, when the network size increases, cunetsim uses additional GPU cores. However,the data transfer between the RAM and the GDRAM remains significant which allows Cunetsim-L4 to be the fastest solution. In both intervals, distributed NS-3 suffers from the initial setup load of the MPI, relegating it behind the classic version.

For large scale, sinalgo has successfully overcome NS-3 thanks to its optimized nodes management. However, the distributed version of NS-3 remains stable overcoming easily sinalgo. On the other hand, cunetsim-L4, reaches the CPU limit while Cunetsim-GPU remains stable in this interval. Finally, for 2000 nodes, both of Cunetsim-GPU and Cunetsim-L4 need 0.35 second, 80 times faster than NS-3 and 26 times faster than sinalgo. For very large scale, the power of cunetsim-GPU is revealed, the number of cores involved in the simulation makes the difference and Cunetsim-L4 fails to follow, even if it remains the most efficient CPU-based simulator. For 48K nodes cunetsim needs 5.93 seconds, 3.5 times faster than cunetsim-L4, 22 times faster than sinalgo and 150 times faster than NS-3. It is interesting to compare in such scale Cunetsim-4L and Distributed NS-3 since both use the same computing power in theory. In fact, Cunetsim-4L overcomes NS-3 due to two major reasons: first it uses a shared memory synchronization (over OpenMP) while NS-3 uses Ethernet (over MPI). Second, The events scheduling is completely different: Cunetsim has a prior knowledge regarding events relationship while NS-3 has only their timestamps as a scheduling information.

From a theoretical point of view, if we suppose that a CUDA core is, as efficient as an i7 core, we can admit that at the same frequency, they are equivalent. As our GPU includes 224 effective CUDA cores @676Mhz and our CPU includes 4 cores @3.6Ghz(overckolecd+ turbo mode), than the maximum theoretical gain is 9.46 which is two times higher than what we achieved. This value suggests that there still exist some interesting optimizations to consider, especially increasing the GPU use which did not exceed 82% while Cunetsim-L4, NS-3 and sinalgo saturate the CPU. The SIMD architecture of the GPU implies that we dedicated the totality of each stream multiprocessor (SM)[1] to 32 nodes until they finished. In such a situation we allocated resources for all nodes, including inactive ones while a sequential execution (CPU) did not waste resources.

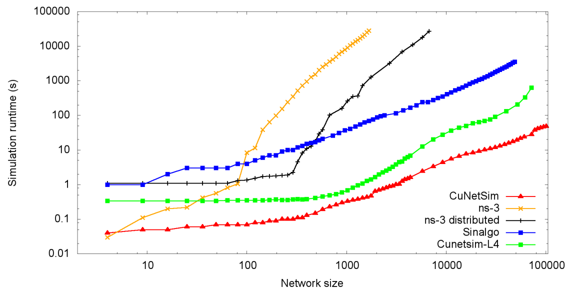


Figure 8. Simulation runtime of the mobile network: the complexity of wireless mobile scenario highlights the limitation of classic approaches under large scale conditions

2) Mobile Scenario: Figure. 8 shows the average measured simulation runtime for each simulator. The mobility imposes the evaluation of nodes connectivity in each round. Once again, the general behavior of the five simulators is similar to the previous scenario. NS-3 is the fastest CPU-based simulator up to 36 nodes and cunetsim-L4 becomes the fastest CPU-based simulator beyond. NS-3 runtime increases exponentially as a function of network size while Sinalgo and Cunetsim-L4 seem more robust. The distributed version of NS-3 increases its leeway but did not influence the global behavior. Thus, distributed NS-3 remains faster than sinalgo up to 800 nodes but its computing time becomes unstable further. Cunetsim-L4 runtime remains relatively invariant for small to medium networks, and becomes a function of nodes' number nearby of 1000. Sinalgo presents a quasi-linear runtime as a function of the network size but cannot achieve a very large scale simulation in realistic time (simulating 48K nodes requires 3552 seconds).

Cunetsim(GPU version) runtime is linear per segment between 4 and 8000 nodes. For each segment the runtime is almost linear. From this threshold, it becomes relative to the network size but remains reasonable, even for $100K$ nodes. In all cases, Cunetsim is extremely faster than all CPU-based simulators. For $48k$ nodes cunetsim is up to 9.2 times faster

[1]Fermi architecture' SMs include 32 Cuda cores

than Cunetsim-L4 and 260 times faster than sinalgo. NS-3 is unable to compete in such scale. In addition the CPU-legacy version presents very interesting results since it is 28 times faster than Sinalgo for the same scale. As the results of the distributed NS-3 prove, distributing the simulation over several machines is not sufficient in itself but must be coupled with, either a clever networking partitioning or a specific distributed event scheduler.

The higher performance of Cunetsim (in both modes) is due to the simultaneous action of four factors.(i)The high parallelism degree of Cunetsim architecture allows efficient use of the GPU computing power and all CPU cores in the CPU-legacy mode. (ii)The connectivity algorithm was designed and optimized to be parallel and distributed taking advantage of the largest cores' number. (iii)The DRAM offers larger bandwidth than the current RAM, theoretically 10 times faster. (iv)The software scheduling task becomes a critical process in CPU context, while its overhead is minimized in GPU since it is achieved using dedicated hardware. Since (iii) and (iv) are not available for the CPU-legacy version, the difference between both versions runtime is growing function of the network size. For a very large network the GPU version is up to 9.2 times faster than the CPU one which proves the interest of using the GPU as a Co-processor.

B. *Memory usage*

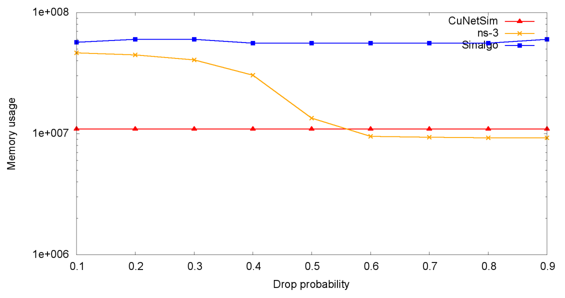


Figure 9. Memory usage vs. Drop probability

Figure. 9 shows the maximum used memory during the simulation as a function of the dropping probability for a fixed network size (3721). We note that both cunetsim versions use exactly the same quantity of RAM then we just mention cunetsim in this section. The drop probability affects the network traffic and the number of exchanged messages. Sinalgo presents a slight decrease of its used memory when the traffic decreases, while NS-3 presents a more flexible behavior and adapts its usage to the network load. Cunetsim uses always the same memory for a fixed network size because each simulated node has at least two fixed buffers. Sinalgo needs between 20% and 600% more than NS-3 while Cunetsim seems more efficient for large traffic load. We notice, however, that NS-3 has a dynamic memory management process while Cunetsim assigns a fixed buffer for each node.

IV. Hardware impact

The performance of cunetsim is directly related to the hardware efficiency, however, the GPU on one hand and the CPU on the other hand affect unequally the simulation performance. This section provides a qualitative and quantitative study of their respective impacts as a function of the number of cores and the frequency. We note that we use the first scenario based on the grid topology(sectionIII) since old devices did not support Curand library.

A. *GPU Impact*

1) Impact of total number of GPU cores: We propose to compare four devices which differ essentially by their number of embedded cores: The 8400GS includes 8 Cuda cores. The FX880M is a mobile GPU including 48 Cuda cores. The GTX 560 includes 336 Cuda cores while the GTX 580 includes 512 one. We varied the network size from 4 to 246K, measured the simulation runtime with each of the four devices and reported the results in Figure. 10. For a small network [4,81], the GPU-CPU data transfer overhead is the bottleneck. For a middle-range network [100, 250], the GPUs having the same architecture offer similar performance. As for networks involving more than 250 nodes, both 8400GS and FX880M GPUs are overloaded by the computation charge and their few cores (8, 48) are no longer able to compete with the GTX 560 and 580 GPUs. The difference between the latter is proportional to their cores' number.

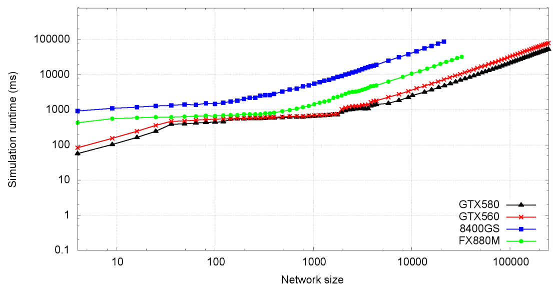


Figure 10. Simulation runtime of different devices: Naturally, more cores means higher efficiency

2) Impact of GPU frequency: For this experiment, we use three devices (1: GTX460@715 Mhz, 2: GTX460 @763 Mhz (reference) and 3: GTX560 @810 Mhz) where the major difference is the GPU frequency. We fixed the network size to 246K nodes and we calculated the runtime of each device. Results are summarized in Figure. 11. The GTX 560 is 7% faster than the reference device while its frequency exceeds by 6.15%. The first device, is 7.5% slower than the reference one while its frequency is lower by 6.71%. These measurements demonstrate that the runtime is proportional to the frequency of the GPU, however, the frequency evolution is generally less significant and more expensive than the Cores' number.

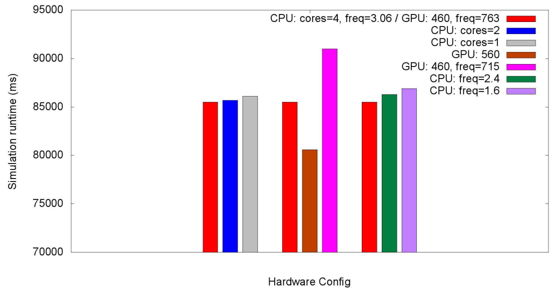


Figure 11. Harware impact for Fully GPU version: neither the CPU frequency nor the number of CPU-embedded cores has a significant impact

B. CPU impact

To evaluate the CPU impact, we distinguish the two versions of cunetsim: The fully-GPU version where the CPU manages the simulation and the CPU-legacy one where it achieves the totality of the simulation.

1) fully GPU: To evaluate the sensibility of the fully GPU version to the CPU capabilities, we fixed the network size to 246K nodes. First we vary the enabled CPU cores from 4 to 1. Second, we reduce its frequency gradually from 3.06 Ghz to 1.6 Ghz. Results are reported on fig 11 where we can observe that the CPU impact is not proportional to the CPU power:reducing the frequency by 45% implies only 5% of performance loss.

2) CPU-legacy version: We conduct 6 series of measurement where we varied the number of threads involved in the simulation between 1 and 8. Our CPU is a i7-920 including four cores with the hyper-threading technology. This means that each core is able to execute two threads. We varied the network size between 4 and $72k$ nodes. Figure.12 summarizes results. As expected, the Cunetsim-L1 runtime is generally the slowest one, Cunetsim-L2 presents a gain of 23 % while the Cunetsim-L4 gain is about 46%. When the number of active threads exceeds the number of physical cores, we observe a relative performance degradation (about 4-5%), this phenomenon is mainly due to concurrence between threads which complicates the scheduling operation.

Based on these results, we conduct 4 series of measurement using the Cunetsim-L4 in the same conditions, where we varied the CPU frequency between 1.6Ghz and 3.6Ghz. Results are reported on Figure.13. We observe that the total runtime is a function of the CPU frequency. In average, An increase of 30% on the frequency, implies an increase of 20% on the performance. These results demonstrate that the legacy version depends on the CPU computing power and profits of the multi-core capability.

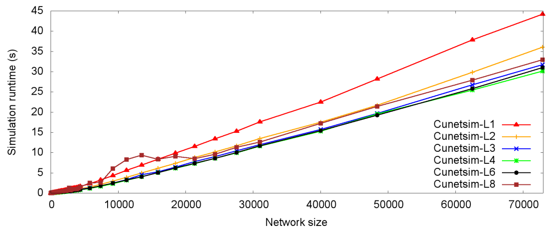


Figure 12. CPU Cores' impact (Legacy): best performance was reached when then number of thread is equal to the number of cores

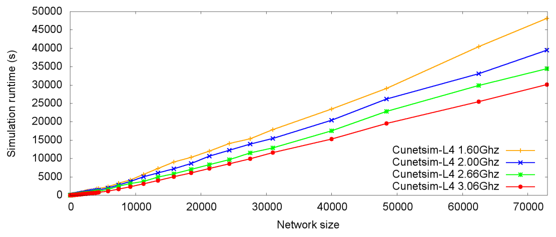


Figure 13. CPU frequency impact (Legacy): As expected, the runtime is linearly related to the CPU frequency

V. Discussion

In this section, we briefly discuss three limiting factors of Cunetsim, namely event scheduler, neighborhood discovering and floating point precision.

Event Scheduler: The conservative approach that we use for WPs in Cunetsim events scheduler presents an efficiency weakness for small network size. To reach a reasonable efficiency ratio, the network size must be greater than the number of GPU cores. Note that current GPU includes up to 1500 cores for mono-GPU devices. On the other hand, this approach can induce an important waste of resources in case of parallelization in CPU context where the number of cores is between 4 and 8 if we address a small network. However, in large scale scenarios, the difference between this schedulers and an optimal one will be reduced.

Neighborhood discovering: Cunetsim implements an optimized connectivity WP which aims to minimize the number of comparisons; the optimization is based on the existence of Rmax. Whatever the wireless technology is, this approach is adapted to the free space model, but remains relevant for terrains which include obstacles and multi-path channel. In this case, the correctness of the simulation will be respected if these variations are taken into account for the calculation of Rmax value. It is worth noting that it remains possible to turn off this optimization at the expense of the simulation runtime.

Floating Point Precision: The implementation of the floating point on NVIDIA device is not fully IEEE compliant. To analyze the difference between a GPU and CPU implementation, we use the distance computing between two nodes as a benchmark test and calculate this distance using both, over 1 million of samples. The difference between each pair of results is less than 0.01%. Depending on the scenario, this difference might cause some simulation inaccuracy (e.g. channel modeling).

VI. Conclusion

New challenges emerge when simulating large scale mobile networks, especially if we consider the paradigm of *very large network* rather than *the network of networks*. While network simulation tools are widely used for validation and performance evaluation, their scalability and efficiency remain challenging. Cunetsim aims to unlock the parallel capabilities of the state-of-the-art hardware and software architectures to achieve simulation scalability and efficiency with significantly lower cost. Cunetsim is the first fully GPU based simulator which provides a CPU-GPU co-simulation framework for large scale scenarios. In contrast with existing GPU acceleration approach, the simulation is fully executed over the GPU. Further, Cunetsim proposes an efficient solution for the management of memory critical section which presents a real challenge of the GPU programming model.

Performance results show that the execution time could be radically improved when GPU parallelism is used to carry out the simulation. In particular, Cunetsim is able to achieve up to 260 faster execution time than existing simulators, when targeting large scale mobile networks. The results also reveal that the existing simulators could be further improved through multi-core parallelism.

Acknowledgments

This paper describes work undertaken in the context of the LOLA and CONECT project. The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement n^o248993 and n^o57616.

References

[1] http://disco.ethz.ch/projects/sinalgo/.

[2] http://www.nsnam.org/.

[3] http://www.pgroup.com/resources/unifiedbinary.htm/.

[4] P. Andelfinger, J. Mittag, and H. Hartenstein. Gpu-based architectures and their benefit for accurate and efficient wireless network simulations. In *MASCOTS, 2011 IEEE 19th International Symposium on*, pages 421–424. IEEE, 2011.

[5] S. Bai and D. Nicol. Acceleration of wireless channel simulation using gpus. In *Wireless Conference (EW), 2010 European*, pages 841–848. IEEE, 2010.

[6] B. Bilel, N. Navid, K. R., and B. C. Openairinterface large-scale wireless emulation platform and methodology. In *MSWIM*. ACM, 2011.

[7] H. Breu and D. Kirkpatrick. Unit disk graph recognition is np-hard. *Computational Geometry*, 9(1):3–24, 1998.

[8] E. Chung, E. Nurvitadhi, J. Hoe, B. Falsafi, and K. Mai. Protoflex: Fpga-accelerated hybrid functional simulator. In *IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[9] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-scale network simulation: how big? how fast? In *11th IEEE/ACM MASCOTS 2003.*, pages 116 – 123, oct. 2003.

[10] J. Harri, F. Filali, and C. Bonnet. Mobility models for vehicular ad hoc networks: a survey and taxonomy. *Communications Surveys & Tutorials, IEEE*, 11(4):19–41, 2009.

[11] C. P. Ken Renard and J. Clarke. A performance and scalability evaluation of the ns-3 distributed scheduler. The Workshop on ns-3 (WNS3)), 2012.

[12] C. Nvidia. Compute unified device architecture programming guide. *NVIDIA: Santa Clara, CA*, 2011.

[13] A. Park and R. Fujimoto. Efficient master/worker parallel discrete event simulation. *2009 ACMIEEESCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 145–152, 2009.

[14] A. Park and R. M. Fujimoto. Parallel discrete event simulation on desktop grid computing infrastructures. *International Journal of Simulation and Process Modelling*, 5(2):157 – 171, 2009.

[15] H. Park and P. A. Fishwick. An analysis of queuing network simulation using gpu-based hardware acceleration. *ACM Trans. Model. Comput. Simul.*, 21(3), Feb. 2011.

[16] K. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 74–81. IEEE Computer Society, 2006.

[17] K. Perumalla, R. Fujimoto, T. McLean, and G. Riley. Experiences applying parallel and interoperable network simulation techniques in on-line simulations of military networks. pages 97–104, 2002.

[18] B. Romdhanne and B. Navid. Cunetsim: a new simulation framework for large scale mobile networks. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, pages 217–219, 2012.

[19] B. romdhanne Bilel and Navid. Cunetsim: A GPU based simulation testbed for large scale mobile networks. In *The 2nd International Conference on Communications and Information Technology (ICCIT): Wireless Communications and Signal Processing (ICCIT-2012 WCSP)*, pages 374–378, June 2012.

[20] E. Weingartner, H. Vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. Ieee, 2009.