Optimization of a peer-to-peer electricity market resolution on GPU

Beatrice Thomas, Roman Le Goff Latimier, Abdelhafid Elouardi, H. Ben Ahmed, Samir Bouaziz

Optimization of a peer-to-peer electricity market resolution on GPU

1st Béatrice THOMAS
Université Paris-Saclay
ENS Paris-Saclay, CNRS, SATIE
91190, Gif-sur-Yvette, France
beatrice.thomas@ens-rennes.fr

2nd Roman LE GOFF LATIMIER
UniR
ENS Rennes, SATIE
35170, Bruz, France
roman.legoff-latimier@ens-rennes.fr

3rd Abdelhafid EL OUARDI
Université Paris-Saclay
ENS Paris-Saclay, CNRS, SATIE
91190, Gif-sur-Yvette, France
abdelhafid.elouardi@universite-paris-saclay.fr

4rd Hamid BEN AHMED
UniR
ENS Rennes, SATIE
35170, Bruz, France
hamid.benahmed@ens-rennes.fr

5rd Samir BOUAZIZ
Université Paris-Saclay
ENS Paris-Saclay, CNRS, SATIE
91190, Gif-sur-Yvette, France
samir.bouaziz@universite-paris-saclay.fr

***Abstract*—The increasing integration of distributed energy resources on the electrical grid will complexify its management by overloading the system operator. One solution explored in the literature is to decentralize the grid management with communities or a peer-to-peer market. However, the computational challenge also applies to researchers when they simulate this management on unique hardware. In this work, extensive research has been done to optimize a simulation of an energy market on a Graphic Processing Unit (GPU). This market will be decentralized using an Alternating Direction Method of Multipliers (ADMM). The new optimized version has enhanced the computation time by about 40% by comparison to the GPU reference method. This allows a resolution of a study case of 2463 agents in 0.15s per hour step or less than one hour for a 3-years simulation.**

***Index Terms*—GPU optimization, Electricity market, Peer to Peer (P2P), Performance evaluation**

I. Introduction

With the increasing integration of renewable energy and active consumers on the electrical grid[1], its management will become even more complex [1]. Having renewable and distributed energy, which has high variability, makes having a real-time market (every 15 minutes) critical. Furthermore, the agents' count's growth may open new ways of grid management while overloading the system operator, which may not be able to find the optimum in time [2]. However, balance at every moment consumption and production is crucial to prevent any collapse of the electrical grid.

Thus, several research studies try to prove that solving the market in a decentralized way using a SmartGrid is possible and relevant [3]. Indeed, by decentralizing the market, two goals are achieved. First, each agent has more privacy because they communicate only their choice. Then, the large-scale problem is divided into many little parallel problems that are easier to solve [4]. A Peer to Peer (P2P) market is decentralized, with no central operator, and every agent can communicate with all the others.

Nevertheless, the decentralized market simulation being often centralized on unique hardware, researchers face prohibitive processing times to realize the proof-of-concept on actual size study cases other than micro-grid. For example, [5] is fast (0.1s) but has only 12 agents in their market, whereas [6] and [7] have more than 30 agents but last respectively 9.5s and 59s (the second adds physical grid constraint). The computational challenge comes when a larger-scale problem (with more than 100 or 1000 agents) is considered.

GPU has proven itself in many domains, such as radars [8] or cartography [9] for the reduction of computation time. This hardware has also been used for power Flow problems [10]- [11] and decentralized electricity market management[2]. Nevertheless, many hardware-specific optimizations must be done to make the most of it.

The objective of this work is multiple. The first is to describe one GPU optimization method to allow optimization on other problems. The second is to give the community an open-source code to solve large-scale P2P markets during long periods to help validate the performances of their management or to check the operation safety, for example.

The remainder of this paper is structured as follows: first, part II presents the algorithm, the hardware architecture, and the optimization methodology for the calculations. Then section III introduces the different optimizations made, and finally, part IV will present the result of these optimizations.

II. Methods and materials

A. Market resolution algorithm

The objective of the peer-to-peer market resolution is to maximize global social Welfare without any central coordinator. That is made via the minimization of the sum of each agent

[1]RTE. Conditions and Requirements for the Technical Feasibility of a Power System with a High Share of Renewables in France Towards 2050

[2]the code of a previous personal study is available here https://gitlab.com/satie.sete/p2p_market_resolution_gpu

$n \in \Omega$ cost function g_n, (1a). This cost function represents the goal of power needed or the cost of producing electricity for each agent. As there is no central agent, each trade t_{nm} between the agents n and m must be considered in addition to the total power p_n of each agent (1c). The other constraints are as follows: what is bought must be sold, (1b), each agent has limits to the exchangeable power (1d)-(1g). According to the agent's type (consumer, generator, or prosumer), the trades must be negative, positive, or between the fixed limits.

The dual variables Λ and μ (Lagrangian multipliers) are respectively associated to the constraints (1b) and (1c).

$$\min_{T,P} \quad \sum_{n\in\Omega} g_n(p_n) \tag{1a}$$
$$\text{s.t.} \quad T = -{}^tT \qquad (\Lambda) \tag{1b}$$
$$p_n = \sum_{m\in\omega_n} t_{nm} \quad (\mu) \qquad n \in \Omega \tag{1c}$$
$$\underline{p_n} \leq p_n \leq \overline{p_n} \qquad n \in \Omega \tag{1d}$$
$$t_{nm} \leq 0 \qquad n \in \Omega_c \tag{1e}$$
$$t_{nm} \geq 0 \qquad n \in \Omega_g \tag{1f}$$
$$\underline{p_n} \leq t_{nm} \leq \overline{p_n} \qquad n \in \Omega_p \tag{1g}$$

To decentralize the resolution, the augmented Lagrangian L_ρ, associated with the minimization and the constraint (1b), is introduced.

$$\begin{aligned} L_\rho(T,P,\Lambda,z) = &\sum_{n\in\omega} g_n(p_n) \\ &+ \sum_{n\in\omega}\sum_{m\in\omega_n} \lambda_{nm}(t_{nm} - z_{nm}) \\ &+ \sum_{n\in\omega}\sum_{m\in\omega_n} \frac{\rho}{2}(t_{nm} - z_{nm})^2 \end{aligned} \tag{2}$$

with z_{nm} the global variable and ρ the penalty factor:

$$z_{nm} = \frac{t_{nm} - t_{mn}}{2} \tag{3}$$

By using a so-called consensus ADMM [12], it can be deduced that solving (1a) and (1b) is the same as solving for each step k and each agent n:

$$\begin{aligned} T_n^{k+1} = &\underset{T_n}{\text{argmin}} \quad g_n(p_n) \\ &+ \sum_{m\in\omega_n} \left(\frac{\rho}{2}(t_{nm} - \frac{t_{nm}^k - t_{mn}^k}{2} + \frac{\lambda_{nm}^k}{\rho})^2\right) \\ &\text{s.t.} \quad (1b) - (1g) \end{aligned} \tag{4}$$

In each step, the Lagrangian multipliers associated with the anti-symmetry constraint are updated as follows:

$$\lambda_{nm}^{k+1} = \lambda_n^{k+1} + \frac{\rho}{2}(t_{nm}^{k+1} + t_{mn}^{k+1}) \tag{5}$$

The following residual calculation for each agent n allows an anticipated termination of the algorithm:

$$r_n^{k+1} = ||t_{nm}^{k+1} + t_{mn}^{k+1}|| \tag{6a}$$
$$s_n^{k+1} = ||t_{nm}^{k+1} - t_{nm}^{k})|| \tag{6b}$$

with $||.||$ being any norm. This part will be called the **global problem** as it concerns all agents.

Per analogy, by using a so-called sharing ADMM [12] with the global variable $z_i = t_i$, it can be deduced that solving (4) and (1c) is the same as solving for each step j and each trade between the agent n and its i^{th} peer:

$$t_i^{j+1} = \underset{t_i}{\text{argmin}} \left(f(t_i) + \frac{\rho_l}{2}||t_i - t_i^j + \overline{t}^j - \overline{z}^j + \mu^j||_2^2\right) \tag{7a}$$
$$\overline{z}^{j+1} = \underset{\overline{z}}{\text{argmin}} \left(g(M_n\overline{z}) + \frac{M_n\rho_l}{2}||\overline{z} - \mu^j - \overline{t}^{j+1}||_2^2\right) \tag{7b}$$

with:

$$f_i(t_i) = \frac{\rho}{2}(t_i - \frac{t_{ni}^k - t_{in}^k}{2} + \frac{\lambda_{ni}^k}{\rho})^2 \tag{8a}$$
$$g(M_n\overline{z}) = g_n(M_n\overline{z}) \tag{8b}$$
$$\mu_i^{j+1} = \mu_i^j + t_i^{j+1} - z_i^{j+1} \tag{8c}$$
$$\overline{z} = mean(z) \tag{8d}$$

All (7) functions are scalar and quadratic ones that can be written in this way:

$$h(x) = \sum_i a_i(x - b_i)^2 + \sum_i c_i \cdot x \tag{9}$$

For the rest, if the coefficients (i.e., a_i, b_i or c_i) refer to the trades' equations, the index will be a t, the index will be p if it relates to the total power equations. Their minimum can be found analytically :

$$x_{min} = \frac{2 \cdot \sum_j a_j \cdot b_j - \sum_j c_j}{2\sum_j a_j} \tag{10}$$

Most of these coefficients are constants and only depend on the agent n considered; thus, they will be defined during the initialization, but the three following coefficients must be updated every step:

$$b_{t1} = 0.5(t_{ni}^k - t_{in}^k) - \frac{\lambda_{ni}^k}{\rho} \tag{11a}$$
$$b_{t2} = t_i^j - \overline{t}^j + \overline{z}^j - \mu^j \tag{11b}$$
$$b_{p1} = \mu^j + \overline{x}^{j+1} \tag{11c}$$

To consider the constraints (1d)-(1g), t_i and $\overline{z}$ are projected on their allowed intervals (respectively $lb_n \leq t_i^{j+1} \leq ub_n$ and $\underline{p_n}/M_n \leq \overline{z}^{j+1} \leq \overline{p_n}/M_n$). To check the convergence of this so-called local problem, the residuals are computed as follows:

$$r^{j+1} = ||\overline{t}^{j+1} - \overline{z}^{j+1}|| \tag{12a}$$
$$s^{j+1} = ||t_i^{j+1} - t_i^j|| \tag{12b}$$

This will be called the **local problem** as each agent can solve its problem independently. The algorithm is presented here in Alg: 1; it is worth noticing that the residual computation will be detailed later.

input : Study case, ρ, ϵ and kmax
output: Vectors T, Λ, residuals

while $err > \epsilon$ *et* $k < kmax$ **do**
  $\Lambda \leftarrow \Lambda + \rho \cdot (T + {}^t T)/2$;
  while $err_l > \epsilon$ *et* $j < jmax$ **do**
    $T^{j+1} \leftarrow \operatorname{argmin} F(T^{j+1})$;
    $T^{j+1} \leftarrow min(T^{j+1}, Ub)$;
    $T^{j+1} \leftarrow max(T^{j+1}, Lb)$;
    $T_{mean} \leftarrow mean(T^{j+1})$;
    $\overline{z} \leftarrow \operatorname{argmin} G(\overline{z})$;
    $\overline{z} \leftarrow min(\overline{z}, \overline{p_n}/M_n)$;
    $\overline{z} \leftarrow max(\overline{z}, \underline{p_n}/M_n)$;
    $\mu \leftarrow T_{mean} + \mu - \overline{z}$;
    if $j\%Step_l == 0$ **then**
      $err1 \leftarrow ||T^{j+1} - T^j||_\infty$;
      $err2 \leftarrow ||T_{mean} - \overline{z}||_\infty$;
      $err_l \leftarrow \max(err1, err2)$;
    end
  end
  if $k\%Step_g == 0$ **then**
    $Res_R \leftarrow ||T - {}^t T||_\infty$;
    $Res_S \leftarrow ||T_k - T_{k-1}||_\infty$;
    $err \leftarrow \max(Res_R, Res_S)$;
  end
end

Algorithm 1: Solving algorithm

B. Software specification

All the code is made with C++ for the Central Processing Unit (CPU) and Cuda for the GPU functions. The object-oriented programming paradigm has been followed. Indeed, classes have been used to represent the study case, the study's parameters, and the study's results, and a personal class has been used to store matrix and their operation on CPU and GPU. One abstract class is used to interface all methods for the calculation. This allows the fast implementation, integration, and testing of new ways of resolution. All definitions and implementation of kernel functions are in this abstract class to prevent compilation errors while allowing the reuse of parts. All have been coded from scratch by the author, and only basic libraries of C++ and Cuda are needed to print messages and measure times. It is worth noticing that all calculations are big-dimension matrix operations; thus, they will be on GPU, and the CPU is used as a controller. The code is split into several functional blocs to allow further analysis, Fig. 1. The meaning of each bloc is resumed in the following table I. The GPU function's calls are depicted with black arrows, and data transfers are shown with blue arrows. Data transfers are reduced to their minimum to increase performance, with one significant transfer at the simulation's beginning and end and just two float transfers at each step.

C. Hardware specification

The algorithm performance evaluation has been made on a GPU GeForce RTX 3060 architecture (Ampere GA 106 architecture). It contains 30 Streaming Multiprocessors (SM) and 3840 Cuda cores. The Roofline of this architecture for the single-precision float is given in the figure 2 according to the benchmark of [13]. On this graph, the measured performance is better than the theoretical one because the GPU can increase its frequency beyond its theoretical limit when the temperature is low. Thus, during the measurement, the frequency of the GPU was higher than the one on the specification. The CPU associated is an AMD RYZEN 5 operating at 3,3GHz.

TABLE I
FUNCTIONAL BLOCKS AND EQUATIONS MATCHING

Block	Description	equations
FB 0	Initialization	
FB 1a	Local problem (B_{t2}, T)	(11b) & (7a)
FB 1b	Local problem (T_{mean})	(8d)
FB 1c	Local problem (B_{p1}, P)	(11c) & (7b)
FB 2	Local residuals	(12)
FB 3a	Global problem (Λ)	(5)
FB 3b	Global problem (B_{t1})	(11a)
FB 4	Global residuals	(6)
FB 5	Results' retrieval	

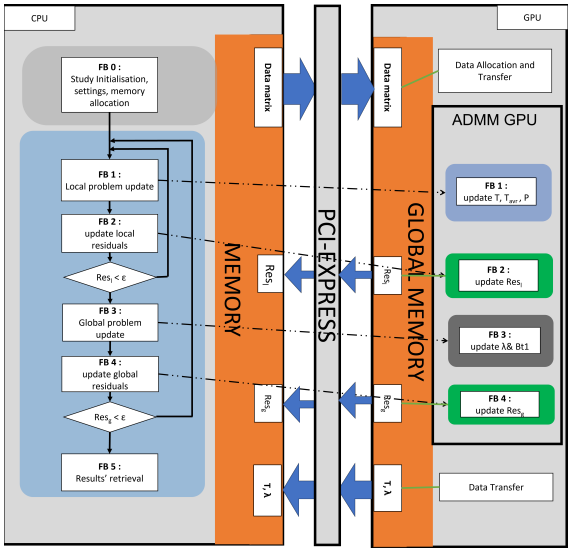

Fig. 1. CPU-GPU partitioning with functional blocs

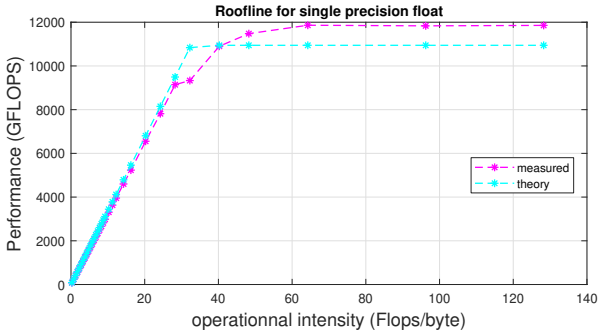

Fig. 2. Measured and theoretical Roofline for single precision float

D. Methodology

To use the hardware-software co-design methodology and check the efficiency of every optimization, the APOD Nvidia optimization methodology[3]: Assert, Parallelize, Optimize, Deploy has been followed. It consists of studying the existing code to find the bottleneck, introduce the possible parallelization, and modify the algorithm to optimize the implementation using the GPU specifications (shared memory, synchronization within a bloc...). The last step is to deploy and benchmark the new version. While it is possible, the whole process can be repeated. Benchmarking is critical because the code run has passed through another black-box architecture-specific optimization process: compilation whose effects are not predictable a priori. The benchmarks will be made using two methods. The first one is based on the micro-benchmarks described by [14] and consists of the processing time measurement of each kernel launch on random data. Copies are made between each call to prevent GPU optimization. As the GPU is asynchronous, barrier functions must be added to measure the time of kernel calls. The mean of ten kernel calls is made one hundred times and stored in a file. This is done with different size of data (*i.e.* different agent's count). The second one, which allows testing the algorithm's behavior, has been made on data from an open database DTU-ELMA/European Dataset [15]. Methods are tested one by one, and the time for the simulation for each hour in a fixed period is measured (with the CPU clock function). Similar tests have been made with barrier functions to measure the time of each functional bloc.

III. Optimization

A. Reference method

The first algorithm version is a trade-off between fast instantiation and processing performance. Indeed, basic optimizations have been used to improve the computation time without requiring extensive studies in opposition to advanced optimizations. The shared optimizations between every method are the following:

- all variables are stored as class attributes, and matrices for intermediate calculations are allocated one time in the beginning to prevent useless allocations and transfers;
- the sparse N^2-sized matrices are transformed into M-sized vector (with N being the agents' count and M the total number of trade);
- all reductions are optimized using [16];
- all calculations are with single-precision float to make the best use of GPU resources;
- the global penalty factor may vary based on residual pairs ratio to make their converging speed equal;
- the residual computation is the infinite norm to prevent error accumulation due to the use of single-precision float;

[3]https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

TABLE II
Optimizations made

Name	Effect
0.a	The study case matrices are copied directly on the GPU instead of duplicated on the CPU and then transferred to the GPU, thus saving one memory copy per matrix
1.a	Each agent uses one bloc of thread, and the three steps are made in one kernel call
1.b	Compute a fixed number of steps in one kernel call
1.c	Use local memory (registers when available)
1.d	Use shared memory
12.a	The two residuals are computed for all agents, which stop at the same time to prevent any divergent branch
12.b	Compute all local steps (with residuals) in one kernel call
3.a	The two phases of this functional bloc are made in the same kernel call
4.a	The two residuals computation are made in the same kernel call

- residuals are not computed every step to reduce the operation's amount and data transfers between CPU and GPU.

This method will be called **GPU5**.

B. Differences between the methods

TABLE III
Methods presentation

Name	GPU6	GPU7	GPU8	GPU9	GPU10	GPU11	GPU12
0.a						✓	✓
1.a	✓	✓	✓	✓	✓	✓	✓
1.b				✓	✓	✓	
1.c				✓			
1.d					✓	✓	✓
12.a	✓		✓	✓	✓	✓	
12.b		✓					✓
3.a	✓	✓					
4.a		✓					

This part will describe the different optimizations used for each functional bloc. Each optimization will be called with one number indicating the functional bloc on which it will be applied and one letter to differentiate several optimizations on the same functional bloc. The optimizations are resumed on the following Table II. To optimize on a GPU, some rules must be known. First, if no *Stream* is used, only one kernel call is active at once, so it is as if there is unnecessary global synchronization between each kernel call. Then, GPU performs better at a high operational intensity (operation count divided by memory usage). However, the reference method's operational intensity is very low (**GPU5**), so the memory access count must be reduced. Finally, GPU has several memories; registers are local to each thread, and the fastest one, shared is for all threads on one block and is faster than the global memory accessible by all threads. Nevertheless, it is worth noticing that if there is not enough space in registers, data will be stored in the so-called local memory at the same place (and

at the same speed) as the global memory. The table II shows the differences between each tested method. The GPU7 way is a naive version of GPU12 that has a memory problem and shows what must not be done to optimize an application on GPU; it is why it isn't considered in the following parts.

IV. Results

This part will show the results of the different tests on the methods. For clarity, only graphs about testing on the ten first days of June 2013 of the European case will be presented. All codes are available on GitLab with the MATLAB script to print the results[4].

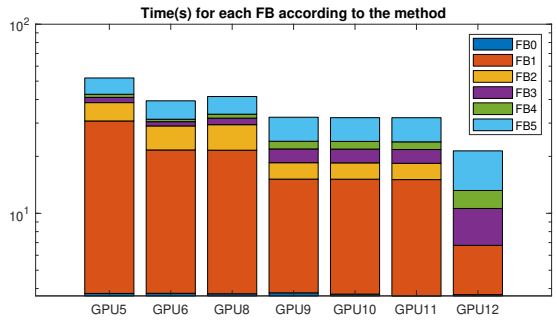

Fig. 3. Measured time for each functional bloc according to the method (240-hours simulation)

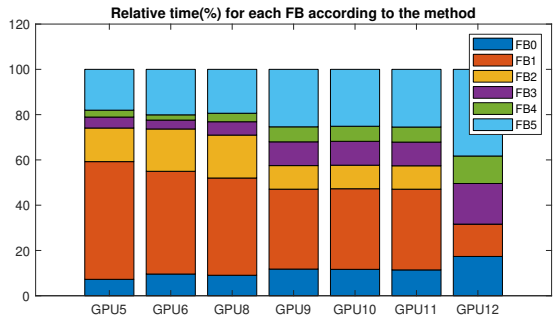

Fig. 4. Relative time for each functional bloc according to the method (240-hours simulation)

First, the two graphs, Fig. 3-4, show the time (respectively the relative time) of each functional bloc according to the method. It is worth noticing that as we add barrier functions, the printed times are far from the real ones when the GPU is asynchronous. Furthermore, readers must be aware of the logarithm scale of the first graph; for instance, the initialization (**FB 0**) lasts around 3s even if it is not visible on the figure. On the first graph, Fig. 3, the improvement of each change is clearly visible. While they are identical, the time variation of some functional blocs shows the variability of the performance. The second figure, Fig. 4, shows that the more computations are optimized, the more the other steps become important.

[4]https://gitlab.com/satie.sete/p2p_market_resolution_gpu

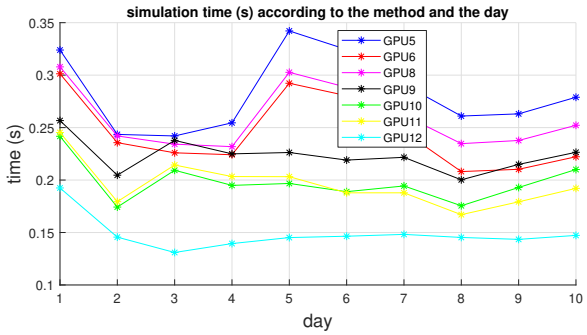

Fig. 5. Average processing time (s) for each pair (method, day) during June 2013

On the graph, Fig. 5, are represented the processing time for each method in the European case for June 2013, grouped by day. Whatever the chosen month, the results are similar; thus, only the result for this month is presented here. It's worth noticing that the time for a one-hour calculation is between 0.1 and 0.5s; these values vary depending on the considered hour and the method used (for reference, optimized CPU methods or non-optimized GPU methods take several seconds on average to process the same thing). The first step processing time is between 1.3 and 2s, which shows what happens without *warm-start*. For each temporal step s, the *SpeedUp* is computed as follows:

$$S_s = \frac{t_{s,GPU5} - t_{s,new}}{t_{s,GPU5}} \tag{13}$$

Results are summarized in the table IV.

TABLE IV
SpeedUp (%) and average time (s) for each method

Method	Avg	Med	Max	Min	Std	T_{avg}
GPU6	12 %	13 %	24 %	- 12 %	0.08	0.24
GPU8	6 %	8 %	15 %	- 13 %	0.06	0.26
GPU9	12 %	16 %	66 %	- 102 %	0.3	0.22
GPU10	23 %	30 %	73 %	- 84 %	0.3	0.20
GPU11	24 %	32 %	74 %	- 86 %	0.3	0.20
GPU12	40 %	44 %	71 %	-5 %	0.2	0.15

It's worth noticing that, on average optimized methods are faster than the basic one (which has $T_{avg} = 0.28s$), but this is not uniform. In some study cases, the basic method outperforms optimized ones. It can also be noticed that making several (or all) steps in one kernel call for the local problem (from **GPU9** to **GPU12**) smooths the average processing time according to the day. Furthermore, using shared rather than local memory is really efficient (**GPU10** against **GPU9**). Nevertheless, reducing the number of matrix copies at the initialization (**GPU11** and **GPU12**) has no visible effect as the complete initialization only happens one time.

V. Conclusion

This study allowed us to increase the time efficiency of an electrical market management algorithm (in the European

case) using a hardware-software co-design method. The time needed to simulate one hour-step has reached 0.15s on average, which allows the simulation of 3 years in less than one hour. The entire implementation is open access on GitLab and can be used by anyone. This will allow solving real scaled problems quicker, hence allowing statistical studies (comparison and variability explanation), dimensioning studies, and stability studies (by adding or removing links or agents)

Achieved performance can be further improved using other architecture having more significant memory blocs, and by changing the software, optimization is applied accordingly. Furthermore, the peer's agent count could be limited to reduce the size of the problem. However, in this case, tests must be done to check the market convergence to the global optimum.

Finally, only the market was considered in this simulation; further investigation could also consider the physical grid constraint.

REFERENCES

[1] Sousa T, Soares T, Pinson P, Moret F, Baroche T, Sorin E. Peer-to-peer and community-based markets: A comprehensive review, Renewable and Sustainable Energy Reviews, Volume 104, 2019, Pages 367-378,ISSN 1364-0321 https://www.sciencedirect.com/science/article/pii/S1364032119300462

[2] Dong A, Baroche T, Le Goff Latimier R, Ben Ahmed, H. Convergence analysis of an asynchronous peer-to-peer market with communication delays. Sustainable Energy, Grids and Networks, 26, 100475. 2021

[3] Garau M, Ghiani E, Celli G, Pilo F, Corti S. Co-simulation of smart distribution network fault management and reconfiguration with LTE communication. Energies, 2018, 11.

[4] Hug G, Kar S, Wu C. (2015). Consensus+ innovations approach for distributed multiagent coordination in a microgrid. IEEE Transactions on Smart Grid, 6(4), 1893-1903.

[5] Sorin E, Bobo L, Pinson P . Consensus-based approach to peer-to-peer electricity markets with product differentiation. IEEE Transactions on Power Systems, 34(2), 994-1004. 2018

[6] Baroche T, Le Goff Latimier R, Pinson P, Ben Ahmed H. Exogenous Cost Allocation in Peer-to-Peer Electricity Markets. IEEE Transactions on Power Systems, Institute of Electrical and Electronics Engineers, 2019, 34 (4), pp.2553 - 2564. ffhal-01964190f

[7] Chernova T, Gryazina E. Peer-to-peer market with network constraints, user preferences and network charges. International Journal of Electrical Power & Energy Systems,Volume 131,2021,106981,ISSN 0142-0615, https://doi.org/10.1016/j.ijepes.2021.106981.

[8] Martelli M, Enderli C, Gac N, Vermesse A, Merigot A. GPU Acceleration: OpenACC for Radar Processing Simulation. 2019 International Radar Conference (RADAR), 2019, pp. 1-6, doi: https://doi.org/10.1109/RADAR41533.2019.171296

[9] Dine A, Elouardi A, Vincke B, Bouaziz S, Speeding up graph-based SLAM algorithm: A GPU-based heterogeneous architecture study. 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP). 2015. pp 72-73, doi: https://doi.org/10.1109/ASAP.2015.7245711

[10] Sooknanan D J, Joshi A . GPU computing using CUDA in the deployment of smart grids. In 2016 SAI Computing Conference (SAI) (pp. 1260-1266). IEEE.2016, July

[11] Araújo I, Tadaiesky V, Cardoso D, Fukuyama Y, Santana Á . Simultaneous parallel power flow calculations using hybrid CPU-GPU approach. International Journal of Electrical Power & Energy Systems, 105, 229-236. 2019

[12] Boyd S, Parikh N, Chu Borja Peleato E, Eckstein J, Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers

[13] Konstantinidis E, Cotronis Y. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing. 2017. vol 107. p 37-56.

[14] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato and M. Püschel. Applying the roofline model. 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014, pp. 76-85, doi: https://doi.org/10.1109/ISPASS.2014.6844463

[15] Jensen T, Pinson P. RE-Europe, a large-scale dataset for modeling a highly renewable European electricity system. Sci Data 4, 170175 .2017. https://doi.org/10.1038/sdata.2017.175

[16] Harris M. NVIDIA Developer Technology. Optimizing Parallel Reduction in CUDA