GPU Techniques for Accelerating Heuristics for Large P-median Clustering Problems

Thé Van Luong and Eric Taillard

1 Introduction

Cluster analysis consists in partitioning a set of entities into subsets (clusters) in such a way that grouped entities of similar kind are into respective categories according to some dissimilarity measures. Numerous models exist for performing such a task. This paper focuses on a model well adapted to data analysis, in the domains of biological networks, medicine, image analysis and text mining. In these domains, the large quantity of data implies to segment them into a relatively large number of small groups that can be then analyzed more deeply with appropriate techniques. The groups are not known a priori. So, an unsupervised clustering model must be used. Moreover, the objects are often not perfectly characterized in these application domains, implying the existence of outliers. In this paper, the model studied is a variant of the p-median problem, that is known to be less sensitive to outliers than the well studied sum-of-squares clustering.

A formal definition of the traditional p-median problem was proposed by Hakimi in [1]. Basically, this problem can be formulated as follows:

$$\text{minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} d(i,j)\cdot x_{ij}$$

subject to

$$(1) \quad \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1,\ldots,n\}$$

$$(2) \quad x_{ij} \leq x_{jj} \quad \forall i,j \in \{1,\ldots,n\}$$

$$(3) \quad \sum_{j=1}^{n} x_{jj} = p$$

$$(4) \quad x_{ij} \in \{0,1\} \quad \forall i,j \in \{1,\ldots,n\}$$

where x_{ij} are binary variables indicating whether or not the entity i is assigned to the center j. This model assumes that each entity can be used as center, that a dissimilarity measure $d(i,j)$ can be computed for each pair (i,j) of entities and that the number of clusters desired is known and equals to p. The objective is to minimize the sum of the dissimilarity measure of all entities with their allocated center. The set of constraints (1) guarantees that each of the n entities is allocated to a single center. The second set of constraints (2) ensures that an entity cannot be

T.V. Luong and E. Taillard are from both HEIG-VD, University of applied sciences of Western Switzerland, Yverdon-les-Bains, Switzerland (e-mail: the-van.luong@heig-vd.ch; eric.taillard@heig-vd.ch)

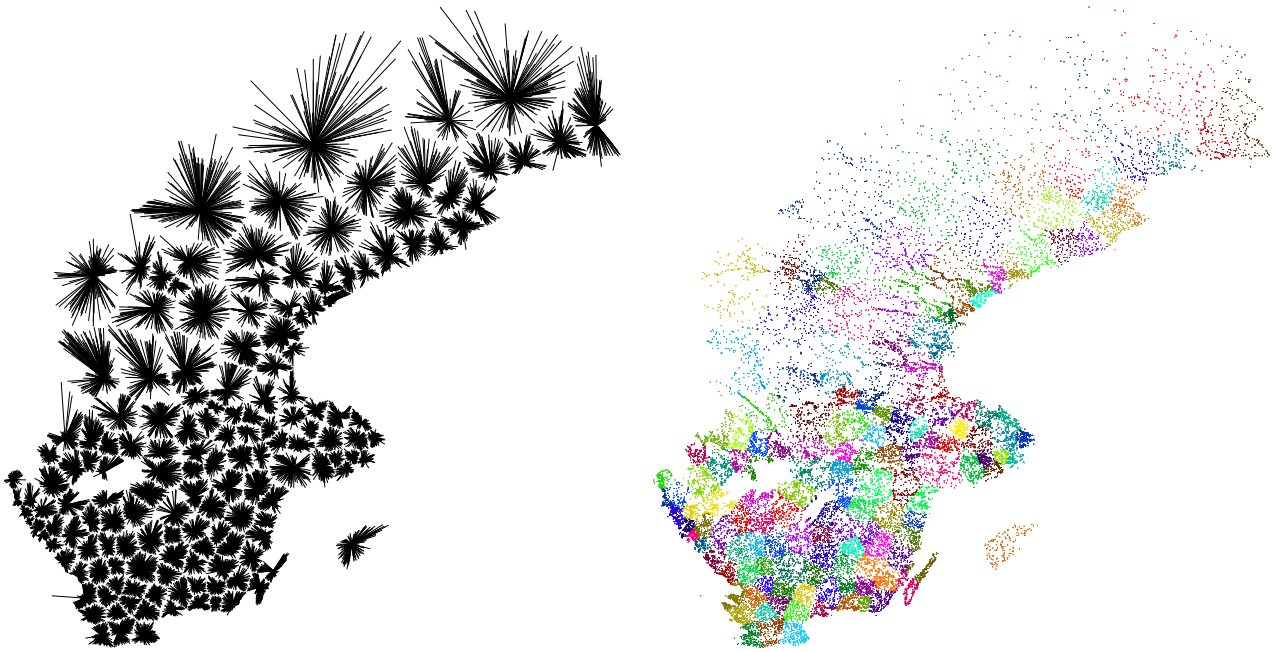

Fig. 1: An example of a solution for the capacitated p-median problem for the Sweden instance. 24978 cities and 158 centers are considered.

allocated to a center that is not opened (the variable x_{jj} indicates if the entity j is considered as a center). The third constraint (3) ensures that exactly p centers are opened.

When dealing with large heterogeneous datasets, the drawback of the traditional p-median model is that it can create very populated clusters in dense regions and clusters with very few entities in sparse regions, even if this tendency is less accentuated than for the sum-of-squares model. To balance the cluster size, a fifth set of constraints (5) can be added to the traditional p-median, leading to the capacitated p-median problem. The last assumes that each entity i has a demand $q_i > 0$ and that the clusters have a capacity Q. This set of constraints can be stated as follows:

$$(5) \quad \sum_{i=1}^{n} q_i \cdot x_{ij} \leq Q \quad \forall j \in \{1, \ldots, n\}$$

However, the capacitated p-median problem is harder to solve than the p-median problem. In the context of data analysis, balancing the cluster size does not mean that each cluster must have exactly the same number of entities. So, set constraints (5) are treated as a soft constraints. A penalty is added to the objective function if a constraint of this set is violated. This means that the constraints of this set can be treated as a redefinition of dissimilarity measure $d(i, j)$, which takes into account the penalty occurring if center j is "overloaded".

Fig. 1 gives an illustration of what a solution could be in p-median problems. At the left side, all the entities are connected with their nearest centers by a line. The right side represents the corresponding clusters.

The p-median problem is one of the most well-known model[1] that has been used in clustering in biological networks [2]. Regarding the space complexity, for large instances, $n \times n$ matrices storing dissimilarity measures are not mandatory since each dissimilarity measure (e.g. an euclidean distance) can be calculated on the fly. Thereby, only linear memory structures ought to be declared and the memory size linearly increases accordingly with the instance size. That

1. It is also known as the k-median problem.

makes the p-median problem an appropriate model when dealing with very large instances of problems commonly found in biology or data mining.

However, for hundred thousands or millions of entities, traditional methods such as the location-allocation procedure are often stopped without convergence being reached. Indeed, in general, the limiting factor in traditional algorithms directly comes from the computational complexity. Thereby, in designing optimization methods, there is often a trade-off between the instance size and the computational complexity to explore it. For instance, while traditional $O(n^2)$ algorithms can solve medium size problems (typically 10^2 to 10^4), $O(n^{3/2})$ methods are a serious alternative to tackle larger problems (typically 10^3 to 10^7). Nevertheless, despite the fact that the complexity is reduced in these methods, the processing time to solve large instances may still be significant and the main process may be worth parallelizing.

Nowadays, the calculation on GPU is recognized as an efficient way to achieve high-performance on long-running scientific applications [3], [4]. Such resources supply a great computing power, are energy-efficient, and unlike grids, they are highly available everywhere: laptops, desktops, clusters, etc. During many years, the use of GPU computing was dedicated to graphics and video applications. Its utilization has recently been extended to other application domains [5], [6] (e.g. scientific computing) thanks to the publication of the CUDA (Compute Unified Device Architecture) development toolkit that allows GPU programming in a C-like language [7].

With the emergence of standard programming languages on GPU and the arrival of compilers for these languages, combinatorial optimization on GPU has generated a growing interest and designing optimization methods are good challenges for GPU computing. The main objective of this report is to highlight GPU techniques that optimization methods could benefit in the case of p-median problems. For doing this, the focus is on the most time-consuming part of numerous advanced clustering methods, namely the location-allocation heuristic introduced by Cooper in [8]. Indeed, it is basically one of the most common and traditional iterative (sub-) heuristic for dealing with centroid clustering problems that must be embedded in advanced heuristics since it ensures, for uncapacitated problems, that the solution produced is stable, i.e. that all entities are allocated to their closest center and all center are placed at their best position in a cluster. This paper discusses step-by-step the parallelization of the location-allocation heuristics and its redesign on GPU accelerators for a variable number of entities and centers.

The rest of the document is organized as follows: Section 2 describes in details each major procedure contained in the general location-allocation heuristic for p-median problems. Section 3 introduces different applications of the location-allocation heuristic that have been used for constructing a solution in $O(n^{3/2})$ in the capacitated p-median problem. In Section 4, parallelization concepts for designing optimization methods on GPU are described. These latter are made in practice in Section 5 for the parallelization of previous $O(n^{3/2})$ methods for p-median problems. Finally, a discussion and some conclusions of this work are drawn in Section 6.

2 Location-allocation Heuristic in P-median Problems

This section presents the location-allocation algorithm which is the foundation of the different methods presented in this paper.

2.1 General Template of the Location-allocation Heuristic

The location-allocation heuristic was first introduced by Cooper in [8] for the multi-source Weber problem. Taillard gives a more refined view of this heuristic for other location-allocation problems in [9]. For p-median problems, a fast variant of the location-allocation procedure is considered in this paper. It is similar to iterative refinement methods that are used in traditional centroid problems. The main body of the location-allocation heuristic is described in Algorithm 1 for the p-median problem.

Algorithm 1 Template for the iterative location-allocation heuristic

Require: n entities, p centers and dissimilarity measures $d(i,j), i,j = 1,\ldots,n$;
1: initialization(n, p, centers, allocated_center);
2: **while** termination_condition not met **do**
3: entities_allocation(n, p, centers, allocated_center);
4: clusters_construction(n, p, allocated_center, clusters, first_position);
5: centers_location(n, p, centers, allocated_center, clusters, first_position);
6: **end while**
Ensure: p centers and clusters are refined.

Basically, in the entities allocation procedure, each of the n entities is allocated to its appropriate center. Thereafter, each cluster is constructed and the number of entities composing the cluster can be determined as well. Thereby, with all these information, centers can be relocated if needed. The process is repeated until a termination condition is reached. This can be for instance when the convergence is reached or when a certain number of iterations has been done. For capacitated problems treated with soft capacity constraints, the convergence might not occurs since the dissimilarity measure may change from one iteration to the next.

Regarding the initialization, p centers are either randomly chosen among the available entities or already determined from previous heuristics.

2.2 Procedures in the Location-allocation Heuristic

After discovering the general template of the location-allocation heuristic, each procedure that has been used in this heuristic must be investigated.

Algorithm 2 provides a template for the entities allocation. Basically, the goal of this procedure is to assign each entity to the center that minimizes the dissimilarity measure. This latter is specific to the p-median problem at hand and it may require additional linear memory structures for its calculation. The complexity of the entities allocation procedure is $O(n \times p)$. Indeed, since one has to allocate each entity to its appropriate center, the dissimilarity measure between each entity and each available center (centers structure) must be calculated. This step is not performed when the corresponding center is the entity itself (lines 2 to 7). The allocated_center structure allows keeping on track the center that has been assigned to each entity.

Next comes the clusters construction procedure (see Algorithm 3). The first_position structure is a linear memory structure that aims at determining the position of each cluster and the number of entities in it. From the allocated_center structure, the number of entities contained in each cluster can be deduced. Thereafter, each cluster can be reconstructed as well (clusters structure). The time complexity of this procedure is in $O(n+p)$. The copy_position structure is an intermediate one that prevents extra calculations on the first_position structure that will be used for the centers location procedure.

Finally, Algorithm 4 provides a template for the centers location procedure. In the latter, it is important to notice that all the clusters have been already constructed (cluster structure). The goal of this procedure is only to relocate the center of each cluster if necessary. The following steps concerns the operations that will be performed on each cluster. As previously seen, the first_position structure delimits the position of each cluster. First at all, the sum of dissimilarities (old_sum variable) between the current center and the other entities in the cluster is calculated (line 2 to line 6). Then, the sum of dissimilarities (new_sum variable) between each potential center and the other entities in the cluster is computed as well (from line 7 to 18). Thereafter, if this sum has been improved, the new center (new_center variable) of the current cluster is relocated via the centers structure (lines 19 and 20).

Algorithm 2 Template for the entities allocation

Require: n entities and p centers;
1: **for** i $\leftarrow$ 0; i $<$ n; i++ **do**
2: j $\leftarrow$ 0;
3: **while** centers[j] $\neq$ i and j $<$ p **do**
4: j++;
5: **end while**
6: **if** j $<$ p **then**
7: allocated_center[i] $\leftarrow$ j;
8: **else**
9: allocated_center[i] $\leftarrow$ 0;
10: **for** j $\leftarrow$ 1; j $<$ p; j++ **do**
11: **if** dissimilarity(i, centers[j]) $<$ dissimilarity(i, centers[allocated_center[i]]) **then**
12: allocated_center[i] $\leftarrow$ j;
13: **end if**
14: **end for**
15: **end if**
16: **end for**
Ensure: All the entities are allocated to their corresponding center.

Algorithm 3 Template for the clusters construction

Require: n entities and their allocated centers;
1: **for** j $\leftarrow$ 0; j $\leq$ p; j++ **do**
2: first_position[j] $\leftarrow$ 0;
3: **end for**
4: **for** i $\leftarrow$ 0; i $<$ n; i++ **do**
5: first_position[allocated_center[i]+1]++ ;
6: **end for**
7: **for** j $\leftarrow$ 0; j $<$ p; j++ **do**
8: first_position[j+1] $\leftarrow$ first_position[j+1] + first_position[j];
9: copy_position[j] $\leftarrow$ first_position[j];
10: **end for**
11: copy_position[p] $\leftarrow$ first_position[p];
12: **for** i $\leftarrow$ 0; i $<$ n; i++ **do**
13: clusters[copy_position[allocated_center[i]]] $\leftarrow$ i ;
14: copy_position[allocated_center[i]]++;
15: **end for**
Ensure: Clusters are constructed and the number of entities of each cluster is known.

Regarding the complexity of the centers location procedure, let n_j be the number of entities that are contained in the cluster j (provided by the first_position structure). Given that $\sum_{j=1}^{p} n_j = n$, a look at the template indicates that line 4 will be performed n times. In a similar manner, line 11 will be repeated at most $\sum_{j=1}^{p} {n_j}^2$ times. This last part is obviously the most time-consuming operation of the procedure. In the best case where all the clusters have around n/p elements, the complexity of the centers location procedure is in $\Omega(n^2/p)$. In the worst case, ($n-p$ entities in one cluster and one entity in the other clusters) the complexity of the procedure is in $O(n^2)$. However, since the clusters are more or less balanced due to soft constraints (5), the complexity

Algorithm 4 Template for centers location

Require: n entities and their allocated centers, p centers and clusters, and the number of entities of each cluster.

```
for j ← 0; j < p; j++ do
  old_sum ← 0;
  for i ← first_position[j]; i < first_position[j+1]; i++ do
    old_sum ← old_sum + dissimilarity(clusters[i],centers[j]);
  end for
  new_center ← centers[j];
  for i ← first_position[j]; i < first_position[j+1]; i++ do
    if clusters[i] ≠ centers[j] then
      new_sum ← 0;
      for k ← first_position[j]; k < first_position[j+1]; k++ do
        new_sum ← new_sum + dissimilarity(clusters[k], clusters[i]);
        if new_sum < old_sum then
          new_sum ← old_sum;
          new_center ← clusters[i];
        end if
      end for
    end if
  end for
  if new_center ≠ centers[j] then
    centers[j] ← new_center;
  end if
end for
```

Ensure: Centers are relocated if necessary.

of the centers location procedure behaves like in the best case.

2.3 Overall Complexity of location-allocation Heuristic

Regarding the complexity of the global location-allocation heuristic, since p is bounded by n, the complexity for the most time-consuming procedures (entities allocation and centers location) is bounded by $O(n^2)$. According to [9], computational experiments showed that the convergence is reached after few hundred iterations. Thereby, since the number of iterations for the convergence of the heuristic is relatively small, it can be considered in $O(1)$. As a result, one can state that the global complexity of the iterative location-allocation heuristic is bounded by $O(n^2)$. Assuming that the clusters are balanced and composed of $\Theta(n/p)$ entities and that the number of iterations of location-allocation is in $O(1)$, the average behavior is more likely close to the best case $\Omega(n \cdot p + n^2/p)$. Therefore, the best complexity is reached when $p = \sqrt(n)$. If the value of p is not in the range of $\sqrt(n)$, the complexity can be maintained to the best complexity with a 2-levels clustering: if p is in $O(n)$ then on can start by building $\sqrt(n)$ (super-) clusters that are each further decomposed into $O(\sqrt(n))$ clusters; if p is in $O(1)$, on can start by building $\sqrt(n)$ (sub-)clusters that are further clustered by treating each of them as a super-entity.

3 Applications of the Location-Allocation Heuristic on the Capacitated P-median Problem

The remaining of the paper relies on the work accomplished by Alvim and Taillard in [10]. In their work, optimization principles are covered to solve large instances of a location routing

(a) Sample (b) Global (c) Small

Fig. 2: Construction of an approximate solution for the capacitated p-median problem. The first phase (sample) consists in finding big clusters of a sample of entities. In the second phase (global), all the available entities are bound to these big clusters. And the final phase (small) concerns the decomposition of these big clusters into small ones.

problem containing million of entities. In this article, a sophisticate construction of an approximate solution with a complexity in $O(n^{3/2})$ is detailed.

The key point of this study is based on solving approximately a p-median problem with capacity for getting an initial solution to the location routing problem. To realize this, the location-allocation procedure presented in the previous section has been widely used during the construction of a solution for the capacitated p-median problem

3.1 Solution Construction based on the Location-allocation Heuristic

The main phases of the resolution of the capacitated p-median problem described in [10] are given in the following. These three phases are also illustrated in Fig. 2.

1) **Sample**: Randomly choose a sample of $30\sqrt{n}$ entities. Solve a relaxation of a capacitated p-median with $p \leftarrow \sqrt{n}$. (Fig. 2(a))
2) **Global**: Assign all n entities to their corresponding centers among the p centers previously found. (Fig. 2(b))
3) **Small**: For $k = 1, \ldots, \sqrt{n}$, decompose each big cluster C_k into $p_k = \lceil \sum_{i \in C_k} q_i / T \rceil$ small clusters by solving a relaxation of a capacitated p-median. (Fig. 2(c))

As quoted above, the different phases for constructing a solution to the capacitated p-median are based on the location-allocation heuristic. Since the instances considered in this article have coordinates in the metric space, the dissimilarity measure $d(i, j)$ corresponds to the distance between entities i and j given by the euclidean distance $d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

The considered relaxation of the capacitated p-median problem is a Lagrangean relaxation that allows taking into account additional constraints of capacity. For the sake of clarity, this step is not detailed in this article. Basically, it consists in adding a penalty function to the distance calculation during the entities allocation procedure. Lagrangean multipliers are updated in an additional procedure according to a gradient method. This alternate procedure may be evaluated in $O(n + p)$.

Since the demands that are considered in this article are variable, T is a parameter of the method such as $T \leq Q$, which allows taking into the consideration the variance of the demands.

TABLE 1: Time measurements in seconds of different parameters for the location-allocation. Only one iteration is considered for each method. A single CPU core implementation on a Intel i7 930 has been used. Four instances of the national and world TSP lib are used for the experiments.

Heuristic	sw24978	ch71009	usa115745	w1904711
Sample	0.02	0.06	0.1	1.5
Global	0.1	0.5	1.1	84.1
Small	0.08	0.2	0.4	9.2
Quadratic	1.0	7.6	19.8	5141.2

Regarding the complexity of the different phases, the sampling of entities consists in calling the location-allocation procedure with $O(\sqrt{n})$ entities and $\sqrt{n}$ centers. Thus, the complexity of this phase is $O(n)$. Regarding the global method, the assignment of all n entities to the nearest $\sqrt{n}$ centers can be implemented in $O(n^{3/2})$ according to the entities allocation and clusters construction procedures presented in Section 2.

The decomposition of big clusters into small ones (small method) consists in executing $O(\sqrt{n})$ times the location-allocation procedure with $O(\sqrt{n})$ entities and $O(\sqrt{n})$ centers. As a result, the complexity of finding $O(n)$ clusters can be done in $O(n^{3/2})$.

One can notice that the different phases of the solution construction have been designed in such a way that the complexity of the overall does not exceed $O(n^{3/2})$.

3.2 Application to National and World TSP Instances for the Capacitated p-median problem

To illustrate this solution construction, experiments have been conducted on the different phases presented above for the capacitated p-median problem.

Four instances of the national and world TSP lib have been considered : Sweden, China, Usa and the world instance. The numbers of cities (entities) are respectively 24978, 71009, 115475 and 1904711. Each city i of TSP instances is represented by its Euclidean position x_i and y_i. Variable demands q_i have been generated using a pseudo-random generator:

$$q_i = (107 \cdot \lfloor |x_i| \rfloor + 97 \cdot \lfloor |y_i| \rfloor) \ mod \ 29 \ + 1$$

The cluster capacity Q has been set to 300 and $T = Q - 10$. Note that during the construction phase of small clusters (small method), the number of centers for each $\sqrt{n}$ big clusters is variable since it heavily relies on the variable demands. According to the pseudo-random generator mentioned above, the number of centers that is observed is $\lceil \sqrt{n}/b \rceil$ where b typically varies between 16 and 20.

The configuration is a 2010 Intel i7 930 using a single CPU core. Regarding the implementation, the code was developed in C++ using the g++ compiler. All the computations involving the Euclidean distances are performed in double precision. In a general manner, in this paper, the focus is on assessing the efficiency of each method. For doing this, the execution time in seconds is measured and all the experiments have been executed 30 times. Since the standard deviation of the results is relatively small and not significant, it is not presented.

A first experiment consists in rapidly showing that $O(n^2)$ algorithms are impracticable for very large instances. To achieve this, each method (i.e. sample, global and small) is compared with a traditional $O(n^2)$ location-allocation procedure. To realize this, n entities and n/b centers have been selected where b varies between 16 and 20 according to the execution (see above). This choice of parameters is made in such way that the complexity of this overall method is quadratic. Table 1 reports the obtained time measurements for this experiment.

TABLE 2: Time measurements in seconds of different parameters for the location-allocation. 300 iterations are considered for the sample and the small phases. A single CPU core implementation on a Intel i7 930 has been used. Four instances of the national and world TSP lib are used for the experiments.

Heuristic	sw24978	ch71009	usa115745	w1904711
Sample	6.4	17.5	28.1	448.8
Global	0.1	0.5	1.1	84.1
Small	3.5	13.2	25.1	1297.1

The first thing that one can notice concerns the quadratic algorithm. The performance difference that occurs between the other methods is quite huge. For example, for the usa115745 instance, the measured time for one iteration is already quite important (i.e. 19s) because hundreds of iterations have to be considered in practice. For the w1904711 instance, due to this large amount of time for one iteration (i.e. more than one hour), it seems clear enough that such an algorithm is hardly practicable even with an implementation on highly-parallel architectures.

Regarding the other methods, in spite of $O(n^{3/2})$ and $O(n)$ complexities, the time spent for the sample and the small phases might become a limiting factor if hundreds of iterations are performed (see Table 2 with 300 iterations). Indeed, this computational time significantly grows with the instance size. Consequently, it is definitely worth parallelizing these methods to speed-up the search process.

4 GPU Computing for Accelerating Optimization Methods

The rapid development of technology in designing processors, networks, and data storage, has made the use of parallel computing more and more popular. Recently, GPU computing has emerged in the recent years as a prominent support to accelerate many complex algorithms [3]. Indeed, most personal computers integrated with GPUs are usually far less powerful than their add-in counterparts. That is the reason why it would be very interesting to exploit this enormous capacity of computing to accelerate optimization methods such as the location-allocation procedure. For doing this, a clear understanding of GPU characteristics is required to provide an efficient implementation of parallel algorithm.

4.1 GPU Architecture

For years, the use of graphics processors was dedicated to graphics applications. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, this architecture provides a tremendous computational horsepower and a very high memory bandwidth compared to traditional CPUs. Fig. 3 illustrates the repartition of components between the two architectures.

One can see that a CPU does not have a lot of arithmetic-logic units (ALU), but a large cache and an important control unit. As a result, the CPU is specialized to manage multiple and different tasks in parallel that require lots of data. Thereby, data are stored within a cache to accelerate its accesses. The control unit will handle the instructions flow to maximize the occupation of arithmetic-logic units, and to optimize the cache management.

Conversely, a GPU has a large number of arithmetic units with a limited cache and few control units. This allows the GPU to calculate in a massive and parallel way the rendering of small and independent elements, while having a large flow of data processed. Since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computations.

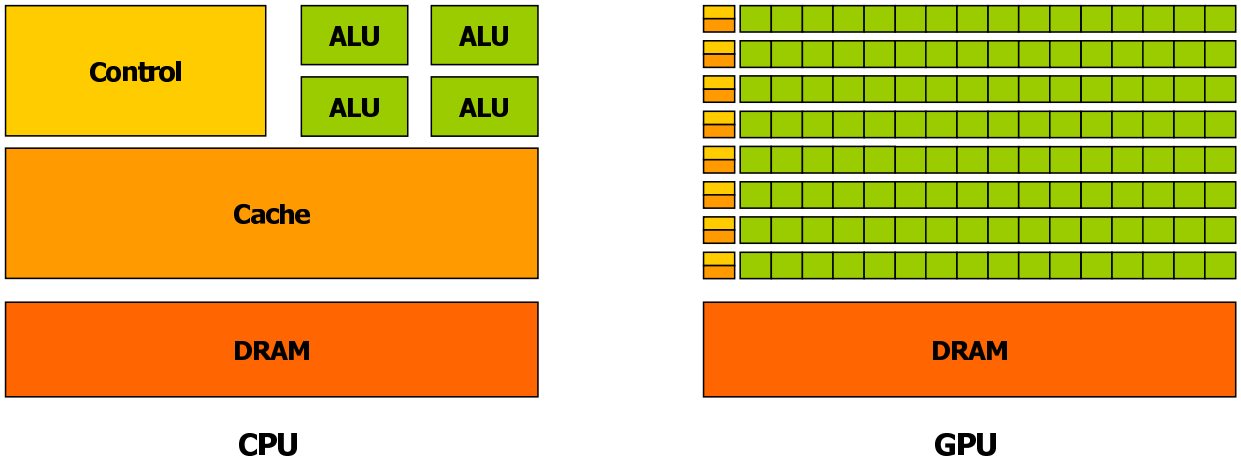

Fig. 3: Repartition of transistors for CPU and GPU architectures.

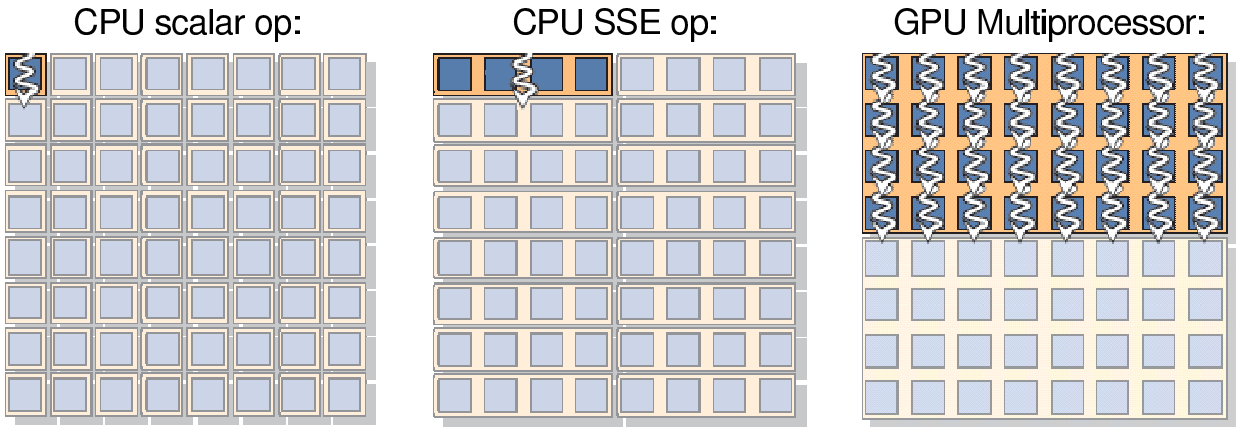

Fig. 4: Comparison of CPU and GPU execution models.

Regarding the execution model, Fig. 4 illustrates it for both CPU and GPU architectures. Basically, a CPU thread proceeds one data element per operation. With the extension of SSE (streaming SIMD execution) instructions, such a CPU thread can operate between two and four data elements. Regarding a GPU multiprocessor, 32 threads proceed 32 data elements. These groups of 32 threads are called *warps*. They are exposed as individual threads but execute the same instruction. Therefore, a divergence in the threads execution provokes a serialization of the different instructions.

4.2 Performance and Issues in GPU-based Algorithms

For the performance obtained in GPU applications, acceleration factors that can be observed are generally within $\times 2$ and $\times 50$ in comparison with a CPU implementation using a single core. Such results are in agreement with the computational power in terms of potential GFLOPS that can be delivered by each device on one modern machine. In other cases, exceptional speed-ups that are claimed in some studies are most of the time due to a comparison with non-optimized CPU implementations using single-precision floating-point operations [11].

In general, the performance difference that occurs in GPU-based algorithms strongly depends if the developed application is compute or memory bound. Parameters to take into account are the problem at hand, the instance size, the time complexity of functions, the number of tasks to evaluate in parallel, the degree of parallelism of the method at disposal, accesses to the global memory, data transfers between the CPU and the GPU, the quality of CPU implementations, and so on.

Parallel combinatorial optimization on GPU is not straightforward and requires a huge effort at design as well as at implementation level. Indeed, several scientific challenges mainly related to the hierarchical memory management have to be achieved. The major issues are the efficient distribution of data processing between CPU and GPU, the thread synchronization, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc. Such issues must be taken into account for the redesign of parallel methods in order to solve large scale optimization problems on GPU architectures. In general, such issues for building efficient methods on GPU are the following:

1) **Cooperation between the CPU and the GPU**. Such a step requires defining the task repartition in the optimization method at hand. For this purpose, the optimization of data transfer between the two components is necessary to achieve the best performance.
2) **Parallelism control**. GPU computing is based on massively parallel multi-threading, and the order in which the threads are executed is not known. Therefore, on the one hand, the degree of parallelism of the method to be parallelized must be high enough. On the other hand, an efficient mapping has to be defined between the entity to parallelize and a thread designated by a unique identifier assigned at GPU runtime.
3) **Memory management**. Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. The different optimization structures have to be placed efficiently on the different memories taking into account their sizes and access latencies.

5 GPU Techniques for Accelerating Location-Allocation Heuristics for p-median Problems

Back to the methods based on the location-allocation procedure presented in Section 3, the goal of this section is to emphasize parallelization techniques and see their contribution in terms of efficiency. Despite the fact that some previous approaches have been proposed for the parallelization of k-means on GPU [12], [13], many parallel optimizations are not easily reusable. Indeed, unlike the p-median problems, the center of each cluster represents the center of gravity. This makes the parallelization more effective since accesses to data structures through the global memory can be drastically minimized in comparison with p-median problems.

5.1 General GPU Model for the Parallelization of Heuristics

Regarding the general location-allocation heuristic, the three main procedures to consider are the entities allocation, the clusters construction and the centers location presented in Section 2. A depth look upon these procedures shows that the potential degree of parallelism is different according to each procedure. As a consequence, some procedures might be completely parallelized on GPU whereas it will remain better to execute some others on CPU. To handle with this, a cooperative approach must be investigated. Fig. 4 illustrates the following GPU model that will be used for the parallelization throughout this section.

Basically, the CPU is considered as a host and the GPU is used as a device coprocessor that is in charge of intensive calculations. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of GPU during the whole execution of the program.

Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple autonomous processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device simultaneously by several processors in parallel.

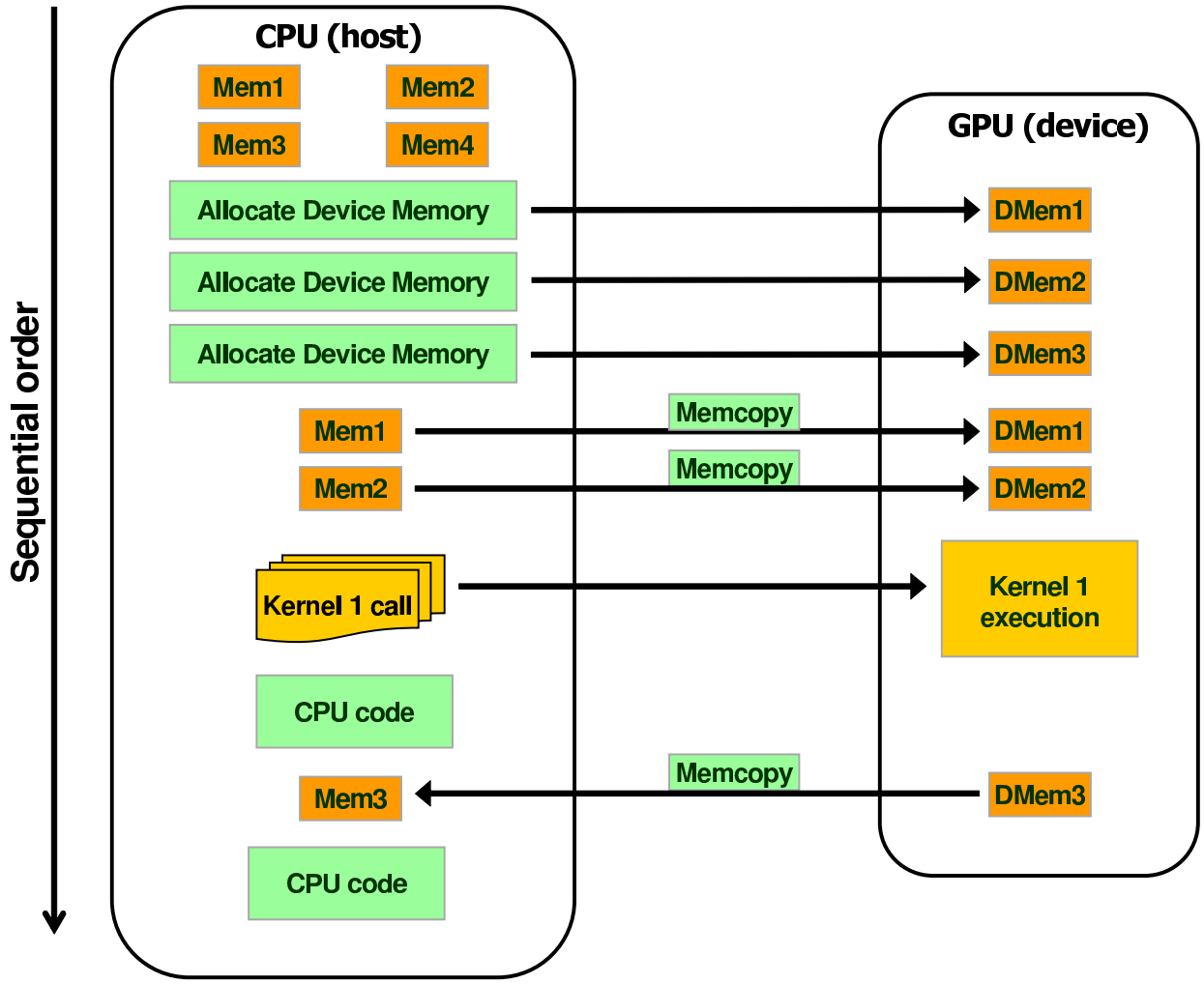

Fig. 5: Illustration of the general GPU model. The GPU can be seen as a coprocessor where data transfers must be performed between the CPU and the GPU.

5.2 Machines Configuration

The different approaches proposed in this document have been experimentally tested on two GPU configurations (see Table 3) using the CUDA toolkit [14]. The GPU cards have different performance capabilities in terms of threads that can be created simultaneously and memory caching. In addition to this, they have a different number of cores which determines the number of active threads being executed. The number of floating point operations per second (GFLOPS) represents a metric for a device performance furnished by vendors.

The first card is a 2010 graphic card based on the Fermi architecture. For the experiments, all the parameter tunings in the CUDA toolkit including L2 memory cache management or the number of threads per block have been made according to this graphic card. Such a fine-grained tuning has not been realized for the second GPU card, which is based on the Kepler architecture. Indeed, such an architecture delivers some additional new functionalities that were not incorporated in the previous generation of cards. Hence, the purpose of using the second configuration is to show that running the same program on a more recent architecture scales quite well.

TABLE 3: Configurations used for the experiments.

Machines	CPU / GPU	GFLOPS	Cores	Memory
Configuration 1	Intel i7 930 2010	49	4 CPU cores	4 GB
	GTX 480 2010	1345	480 GPU cores	1.5 GB
Configuration 2	Intel i7 3770K 2012	125	4 CPU cores	16 GB
	GTX Titan 2013	4500	2688 GPU cores	6 GB

5.3 Application of a Simple Parallelization Scheme on GPU

From the previous parameters used in Section 2, a quick profiling of the sample phase for the world instance on the first configuration with 300 iterations has been performed. The result indicates that the entities allocation procedure nearly occupies 95% of the total execution time. Therefore, a first natural strategy to speed-up the location-allocation heuristic is to only parallelize the entities allocation on GPU. Indeed, in this configuration for the sample phase, the maximal acceleration that can be expected is around $\times 20$, which is pretty significant. Algorithm 5 provides the general template for the parallelization of the location-allocation heuristic on GPU. Lines highlighted in bold indicate major adaptations that must be done for a GPU implementation.

Algorithm 5 Template for the simple parallelization scheme on GPU in the location-allocation heuristic.

Require: n entities, p centers and dissimilarity measures $d(i,j), i,j = 1,\ldots,n$;

1: initialization(n, p, centers, allocated_center);
2: **GPU_memory_allocation(n, p, d_centers, d_allocated_center);**
3: **CPU_to_GPU_copy(n, p, centers, allocated_center, d_centers, d_allocated_center);**
4: **while** termination_condition not met **do**
5: **GPU_entities_allocation_kernel(n, p, d_centers, d_allocated_center)**;
6: **GPU_to_CPU_copy(n, allocated_center, d_allocated_center);**
7: clusters_construction(n, p, allocated_center, clusters, first_position);
8: centers_location(n, p, centers, allocated_center, clusters, first_position);
9: **CPU_to_GPU_copy(p, centers, d_centers);**
10: **end while**

Ensure: p centers and clusters are refined.

First of all, at initialization stage, memory allocations on GPU are made (line 2) on device structures: centers and the allocated_center structure (prefixed by a $d\_$ for representing structures allocated on GPU device memory). Additional structures which might facilitate the computation of the main procedure might be allocated on GPU as well. Second, once the memory allocation is done, all the necessary structures on CPU have to be copied into their counterparts on the GPU memory (line 3). Third, comes the entities allocation kernel where the parallelization is realized on GPU. To understand the concept of kernel, Algorithm 6 illustrates what such a template could look like.

The differences in comparison with the sequential entities allocation presented in section 2 are highlighted in bold. For this procedure, the parallelization is relatively straightforward. Indeed, since the allocation of each entity to its appropriate center is an independent task, a parallelization operated on the entities processing seems quite natural. For doing this, in the corresponding kernel, each GPU thread characterized by a unique *id* is associated with each entity and the kernel is executed with n threads. The instructions that are represented in this template exactly corresponds to the instructions that will be executed by each thread in parallel.

Algorithm 6 Template for the entities allocation kernel on GPU

Require: n entities and p centers **previously defined on GPU device memory**;
1: **i** $\leftarrow$ **thread_id()**;
2: j $\leftarrow$ 0;
3: **while** centers[j] $\neq$ i and j $<$ p **do**
4: j++;
5: **end while**
6: **if** j $<$ p **then**
7: allocated_center[i] $\leftarrow$ j;
8: **else**
9: allocated_center[i] $\leftarrow$ 0;
10: **for** j $\leftarrow$ 1; j $<$ p; j++ **do**
11: **if** dissimilarity(i, centers[j]) $<$ dissimilarity(i, centers[allocated_center[i]]) **then**
12: allocated_center[i] $\leftarrow$ j;
13: **end if**
14: **end for**
15: **end if**
Ensure: All the entities are assigned to their allocated center **on GPU device memory**;

TABLE 4: Measures in terms of efficiency of the **sample method** on GPU. The parallelization strategy is made on the entities allocation procedure. Four instances of the national and world TSP lib in the capacitated p-median problem are considered.

Instance	Intel i7 930 2010 GTX 480 2010		Intel i7 3770K 2012 GTX Titan 2013	
	CPU	GPU	CPU	GPU
sw24978	6.4	$1.0_{\times 6.3}$	3.4	$0.5_{\times 7.4}$
ch71009	17.5	$1.6_{\times 10.9}$	9.6	$0.8_{\times 12.0}$
usa115475	28.1	$2.2_{\times 12.8}$	15.5	$1.0_{\times 15.1}$
w1904711	448.8	$26.6_{\times 16.9}$	252.4	$8.6_{\times 29.3}$

Back to the previous template for the parallelization of the sample heuristic, once the entities have been allocated to their corresponding centers on GPU, the structure representing n allocated centers has to be copied from GPU to CPU (line 6). Intermediate structures that are not defined in this template might be recopied as well. Thereafter, the hand is returned back to the CPU for doing a sequential construction of clusters, then the associated centers location (lines 7 and lines 8). Finally, when these operations are accomplished, p centers must be copied back to the GPU device memory for the next iteration. The process is repeated until a certain condition has been met (e.g. a certain number of iterations).

Regarding the implementation of the previous concepts, the sample method is the first one to be parallelized ($30\sqrt{n}$ entities and $\sqrt{n}$ centers). Table 4 reports the obtained results of a GPU implementation for the national and world TSP lib.

We can clearly see the benefits of parallelizing the entities in the entities allocation procedure. Indeed, for the first configuration, the obtained speed-up in comparison with the associated CPU implementation grows with instance size. It varies between $\times 6.3$ and $\times 16.9$. Regarding the second configuration, similar accelerations are obtained as well for the three first instances. The best performance is achieved for the world instance where the speed-up reaches $\times 29.3$. As a matter of fact, one can confirm that the implementation seems to scale quite well for this instance.

TABLE 5: Measures in terms of efficiency of the **global method** on GPU. The parallelization strategy is made on the entities allocation procedure. Four instances of the national and world TSP lib in the capacitated p-median problem are considered.

Instance	Intel i7 930 2010 GTX 480 2010		Intel i7 3770K 2012 GTX Titan 2013	
	CPU	GPU	CPU	GPU
sw24978	0.1	< 0.01	0.06	< 0.01
ch71009	0.5	< 0.01	0.2	< 0.01
usa115475	1.1	$0.02_{\times 58.9}$	0.6	$0.02_{\times 28.2}$
w1904711	84.1	$1.1_{\times 76.6}$	41.8	$1.0_{\times 43.1}$

TABLE 6: Measures in terms of efficiency of the **small method** on GPU. The parallelization strategy is made on the entities allocation procedure. Four instances of the national and world TSP lib in the capacitated p-median problem are considered.

Instance	Intel i7 930 2010 GTX 480 2010		Intel i7 3770K 2012 GTX Titan 2013	
	CPU	GPU	CPU	GPU
sw24978	3.5	$0.9_{\times 4.0}$	1.3	$0.9_{\times 1.4}$
ch71009	13.2	$2.1_{\times 6.4}$	5.8	$1.6_{\times 3.6}$
usa115475	25.1	$2.4_{\times 10.3}$	11.9	$1.6_{\times 7.7}$
w1904711	1297.2	$100.0_{\times 13.0}$	691.2	$110.8_{\times 6.2}$

Regarding the other instances, since the second graphic card is more recent, one could have expected better accelerations. However, as previously said, fine-grained tunings such as L2 memory cache management has not been done on the second configuration. Indeed, since it is a new architecture, algorithms might be re-thought to obtain further performance. Moreover, it may be difficult to compare the performance just only based on relative acceleration factors, since the CPU implementation for the second configuration is already much faster than the first one.

The second method for the experiments that has been considered concerns the global phase (n entities and $\sqrt{n}$ centers). All the results for the same parallelization scheme for the global method are depicted in Table 5. Acceleration factors for the two first instances are not reported since the computational time of the two GPU implementation is not significant (less than a hundredth second).

The first thing that one can notice concerns the obtained acceleration factors that are quite impressive. In comparison with the previous experience such a performance improvement can be explained by the consequent amount of calculations that can take advantage of parallelism. Since a higher number of entities is available (n threads instead of $30\sqrt{n}$ threads), the degree of parallelism will be higher as well. Indeed, one of the key points to achieve high performance is to keep the GPU multiprocessors as active as possible. This is what is done here when trying to maximize hardware utilization.

A first conclusion of these experiments indicates that a simple parallelization scheme seems particularly efficient to deal with very large instances, especially when the degree of parallelism of the method at hand is important.

Finally, Table 6 depicts the obtained results for the parallelization of the sample phase ($O(\sqrt{n})$ entities and $O(\sqrt{n})$ centers per big cluster). The $\sqrt{n}$ sub-problems have been implemented in such a way that all the independent sub-problems are executed in parallel at the same time.

One can clearly see for the second configuration that the parallelization does not scale with the augmentation of the number of processors and the instance size. This is simply due to the fact

that for the small phase, the entities allocation part is not the only significant operation of the whole algorithm. Hence, all these observations highlight the limitation of a simple parallelization scheme.

5.4 Application of an Improved Parallelization Scheme on GPU

As shown above, a basic GPU parallelization scheme might be really efficient for $O(n^{3/2})$ procedures involving a large number of entities. This probably does not hold when dealing with more sophisticate procedures, in which looking for further improvements might be relevant.

Regarding the potential improvements that can be made, a natural way of proceeding would be to parallelize the two other procedures of the location-allocation heuristic. Nevertheless, a look upon the template for the clusters construction (see Algorithm 3) indicates that this procedure is not really prone to parallelization. Indeed, on the one hand, it is not easy to figure out if the threads parallelization should be focus either on entities or centers. On the other hand, one may notice that in some structures such as the first_position structure, values are incrementally updated in a dependent manner. That makes the parallelization hard to achieve for such a small potential performance improvement.

Conversely, for the centers location procedure, if one refer to Algorithm 4 presented in Section 2, in a similar way than the entities allocation, a natural idea would be to parallelize the centers processing. Unfortunately, in a general manner, since the number of centers is relatively small in comparison with the number of entities, the degree of parallelism may be limited. Indeed, the number of GPU threads that will be created at runtime may not be enough to cover the memory access latency. This is particularly true with $\sqrt{n}$ centers, which only represents 1380 centers for the world instance. As a consequence, if the number of centers is not high enough, such an additional parallelization is likely to provoke a decrease of the overall performance.

As a matter of fact, when designing a parallel method on GPU, one should investigate about the best way to obtain the maximum degree of parallelism. Back to the centers location procedure, there is actually an efficient way to parallelize it. If one pay more attention to the template of the centers location (see Algorithm 4), the main part of this procedure is executed in this following order: **for each cluster**, the sum of dissimilarities of each potential center with the other entities of the cluster is calculated. In fact, one can observe that it can be rewritten in another way that changes the degree of parallelism: **for each entity**, the sum of dissimilarities of this potential center with the other entities of the cluster is calculated. Such a technique of algorithm rewriting is likely to improve the performance of programs in parallel environments.

Algorithm 7 provides the new template for the parallelization of the location-allocation heuristic on GPU. The lines in bold indicate the new changes in regards with the previous version. Basically, the initial centers location procedure is split into two cooperative procedures. In this way, the most parallelizable part is performed on GPU, whereas the sequential and dependent one is done on CPU.

Regarding the part to perform in parallel on GPU, the associated kernel is exclusively associated with the calculation of the sum of dissimilarities in a same cluster. This is typically the most time-consuming part of the centers location procedure. Algorithm 8 provides the template of the corresponding kernel. One can notice the sum structure that is used to store the calculated sums of dissimilarities.

Regarding the new template of the centers location, it is similar to the previous one, except the fact that sums of dissimilarities have been already calculated on GPU through the sum structure previously mentioned.

A new experiment consists to assess the impact of such a new parallelization strategy for the sample method. Table 7 depicts the new obtained results.

The new column corresponds to the latest parallelization scheme. Whatever the used configuration, one can observe a significant improvement for the three first instances where the entities

Algorithm 7 Template for the second parallelization scheme on GPU concerning the centers location in the location-allocation heuristic.

Require: n entities, p centers and dissimilarity measures $d(i,j), i,j = 1,\ldots,n$;
initialization(n, p, centers, allocated_center);
GPU_memory_allocation(n, p, d_centers, d_allocated_center, **d_clusters**, **d_sum**);
CPU_to_GPU_copy(n, p, centers, allocated_center, d_centers, d_allocated_center);
while termination_condition not met **do**
  GPU_entities_allocation_kernel(n, p, d_centers, d_allocated_center);
  GPU_to_CPU_copy(n, allocated_center, d_allocated_center);
  clusters_construction(n, p, allocated_center, clusters, first_position);
  CPU_to_GPU_copy(n, allocated_center, clusters, d_allocated_center, d_clusters);
  GPU_sum_calculation_kernel(n, p, d_allocated_center, d_clusters, d_sum);
  GPU_to_CPU_copy(n, sum, d_sum);
  centers_location2(n, p, centers, allocated_center, clusters, first_position, sum);
  CPU_to_GPU_copy(p, centers, d_centers);
end while
Ensure: p centers and clusters are refined.

Algorithm 8 Template for the sum calculation kernel on GPU

Require: n entities and their allocated centers, p clusters, and the number of entities of each cluster previously defined on GPU device memory;
i $\leftarrow$ thread_id();
sum[i] $\leftarrow$ 0;
for j $\leftarrow$ first_position[allocated_center[i]]; j $<$ first_position[allocated_center[i]+1]; j++ **do**
  if clusters[j] $\neq$ i **then**
    sum[i] $\leftarrow$ sum[i]+ dissimilarity(i, clusters[j]);
  end if
end for
Ensure: All the sums of dissimilarities are calculated on GPU device memory;

TABLE 7: Measures in terms of efficiency of the **sample method** on GPU. The parallelization strategy on the centers location is added to the previous parallelization scheme. Four instances of the national and world TSP lib in the capacitated p-median problem are considered.

Instance	Intel i7 930 2010 GTX 480 2010			Intel i7 3770K 2012 GTX Titan 2013		
	CPU	GPU	GPU2	CPU	GPU	GPU2
sw24978	6.4	$1.0_{\times 6.3}$	$0.4_{\times 16.8}$	3.4	$0.5_{\times 7.4}$	$0.3_{\times 10.2}$
ch71009	17.5	$1.6_{\times 10.9}$	$0.9_{\times 20.0}$	9.6	$0.8_{\times 12.0}$	$0.5_{\times 17.6}$
usa115475	28.1	$2.2_{\times 12.8}$	$1.2_{\times 24.2}$	15.5	$1.0_{\times 15.1}$	$0.7_{\times 23.3}$
w1904711	448.8	$26.6_{\times 16.9}$	$22.3_{\times 20.1}$	252.4	$8.6_{\times 29.3}$	$7.6_{\times 33.2}$

TABLE 8: Measures in terms of efficiency of the **small method** on GPU. The parallelization strategy on the centers location is added to the previous parallelization scheme. Four instances of the national and world TSP lib in the capacitated p-median problem are considered.

Instance	Intel i7 930 2010 GTX 480 2010			Intel i7 3770K 2012 GTX Titan 2013		
	CPU	GPU	GPU2	CPU	GPU	GPU2
sw24978	3.5	$0.9_{\times 4.0}$	$0.5_{\times 7.6}$	1.3	$0.9_{\times 1.4}$	$0.3_{\times 4.8}$
ch71009	13.2	$2.1_{\times 6.4}$	$1.3_{\times 10.1}$	5.8	$1.6_{\times 3.6}$	$0.8_{\times 7.0}$
usa115475	25.1	$2.4_{\times 10.3}$	$2.2_{\times 11.3}$	11.9	$1.6_{\times 7.7}$	$1.3_{\times 9.1}$
w1904711	1297.2	$100.1_{\times 13.0}$	$50.0_{\times 25.9}$	691.2	$110.8_{\times 6.2}$	$34.1_{\times 20.2}$

allocation is not the only dominating operation. For instance, for the first configuration, the previous speed-ups varied between $\times 6.3$ to $\times 12.8$ whereas the new ones alternate from $\times 16.8$ to $\times 24.2$.

Regarding the small method (see Table 8), the benefits of the new parallelization scheme are also quite visible. This is particularly the case for the world instance, where the acceleration factor has been significantly improved. Indeed, they vary from $\times 13$ to $\times 25.9$ for the first configuration and from $\times 6.2$ to $\times 20.2$ for the second one.

6 Discussion and Conclusion

As well as sum-of-squares clustering, the p-median problem has been recognized as a leading model in clustering in many fields such as biology or data mining. However, traditional methods such as the allocation-location procedure might take a prohibitive computational time when dealing with very large instances.

To deal with this, parallel optimization allows improving the effectiveness and robustness of methods in combinatorial optimization. Their exploitation for solving very large problems is possible only by using important computational resources. High performance computing based on the use of GPUs has been revealed to be a good way to provide such computational power. However, the exploitation of parallel models is not trivial and many issues related to the GPU execution have to be considered.

This research report investigates on different parallelization techniques for location-allocation based procedures for p-median problems. In the proposed strategies, the CPU manages the sequential process and let the GPU be used as a coprocessor dedicated to intensive calculations. In particular, the methodology enables to gain up significant acceleration factors compared with a single core architecture.

A perspective of this work is to investigate the scalability of developed parallel techniques for different methods by increasing the number of processors used. In particular, to experimentally validate if it is possible to solve problems in $O(n)$ using $O(\sqrt{n})$ processors.

Furthermore, GPU-accelerated algorithms designed in this report only exploit a single CPU core. In some cases, it may become valuable to fully utilize the other remaining CPU resources. It is particularly significant when the acceleration factors obtained by the GPU-based algorithm are relatively modest. Indeed, all processors are nowadays multi-core and when coupled with GPU devices, performance of GPU-based optimization methods might be significantly improved.

References

[1] S. L. Hakimi, "Optimum distribution of switching centers in a communication network and some related graph theoretic problems," *Operations Research*, vol. 13, no. 3, pp. pp. 462–475, 1965. [Online]. Available: http://www.jstor.org/stable/167810

[2] F. Hüffner, R. Niedermeier, and S. Wernicke, "Fixed-parameter algorithms for graph-modeled data clustering," in *Clustering Challenges in Biological Networks*. World Scientific, 2009, pp. 3–28.

[3] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, "Program optimization carving for gpu computing," *J. Parallel Distribributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.
[4] J. D. Owens, M.Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.
[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.
[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[8] L. Cooper, "Location-allocation problems," *Operations Research*, vol. 11, no. 3, pp. pp. 331–343, 1963. [Online]. Available: http://www.jstor.org/stable/168022
[9] É. D. Taillard, "Heuristic methods for large centroid clustering problems," *J. Heuristics*, vol. 9, no. 1, pp. 51–73, 2003, old technical report IDSIA-96-96. [Online]. Available: http://mistic.heig-vd.ch/taillard/articles.dir/Taillard2003JOH.pdf
[10] A. C. F. Alvim and É. D. Taillard, "Popmusic for the world location routing problem," *EURO Journal on Transportation and Logistics*, 2012, accepted for publication. [Online]. Available: http://mistic.heig-vd.ch/taillard/articles.dir/AlvimTaillard2012.pdf
[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ISCA*, 2010, pp. 451–460.
[12] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-means algorithm by gpus," *J. Comput. Syst. Sci.*, vol. 79, no. 2, pp. 216–229, 2013.
[13] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via cuda," in *Proceedings of the 2009 First International Conference on Intensive Applications and Services*, ser. INTENSIVE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 7–15. [Online]. Available: http://dx.doi.org/10.1109/INTENSIVE.2009.19
[14] NVIDIA, *CUDA Programming Guide Version 5.0*, 2013.