Nearest Neighbors Search Using Multi-GPU

Vinícius Nogueira[1], Lucas Amorim[1,2], Igor Baratta[3], Gabriel Pereira[1], Renato Mesquita[3]

[1]Department of Computer Science
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

[2]Departamento de Computação e Construção Civil
Federal Center for Technological Education of Minas Gerais (CEFET-MG)
Timóteo – MG – Brazil.

[3]Graduate Program in Electrical Engineering
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{viniciusnogueira,gabrielfelipe}@dcc.ufmg.br,
{lucaspa,igorbaratta}@gmail.com, renato@cpdee.ufmg.br

Abstract. *Meshless methods are increasingly gaining space in the study of electromagnetic phenomena as an alternative to traditional mesh-based methods. One of their biggest advantages is the absence of a mesh to describe the simulation domain. Instead, the domain discretization is done by spreading nodes along the domain and its boundaries. Thus, meshless methods are based on the interactions of each node with all its neighbors, and determining the neighborhood of the nodes becomes a fundamental task. The k-nearest neighbors (kNN) is a well-known algorithm used for this purpose, but it becomes a bottleneck for these methods due to its high computational cost. One of the alternatives to reduce the kNN high computational cost is to use spatial partitioning data structures (e.g., planar grid) that allow pruning when performing the k-nearest neighbors search. Furthermore, many of these strategies employed for kNN search have been adapted for graphics processing units (GPUs) and can take advantage of its high potential for parallelism. Thus, this paper proposes a multi-GPU version of the grid method for solving the kNN problem. It was possible to achieve a speedup of up to* $1.99\times$ *and up to* $3.94\times$ *using two and four GPUs, respectively, when compared against the single-GPU version of the grid method.*

1. Introduction

Meshless methods are increasingly gaining space in the study of electromagnetic phenomena as an alternative to traditional mesh-based methods such as the finite element method (FEM) [Amorim et al. 2019, Kamranian et al. 2017, Liu 2016, Jin 2015, Garg et al. 2015, Liu 2002]. The presence of a mesh in the FEM simplifies the computation of the linear equations system, since node connectivity can be obtained directly from it. However, the quality of the mesh affects not only the approximation error of the finite element solution but also the spectral characteristics of the corresponding linear system [Du et al. 2009]. In 2D problems discretized using triangular meshes, ensuring

satisfactory mesh quality is attainable, and many open-source packages are available, such as GMSH [Geuzaine and Remacle 2009]. However, achieving good quality meshes with moderate computational resources for arbitrary three-dimensional problems is still an open problem. This problem gets even more complicated if we consider moving domains, requiring re-meshing at each step, which can even make the adoption of mesh-based methods unfeasible [Chen et al. 2017].

In the meshless methods these problems no longer exist. The domain discretization is done by spreading nodes along its domain and boundaries, and a system of algebraic equations is established without the use of a mesh. Due to the lack of connectivity information, additional computational steps are required for computing neighborhood information (node-node interactions), which can lead to an additional computational cost in contrast to mesh-based methods.

Meshless methods, such as the Meshless Time-Domain Method (MTDM) [Ikuno et al. 2013] and Meshless local Petrov-Galerkin method (MLPG) [Amorim et al. 2019], have been used to solve wave propagation and static problems, respectively. In these methods, different shape functions are used, such as the Radial Point Interpolation Method (RPIM), Radial Point Interpolation Method with polinomial reproduction (RPIMp) and Moving Least Squares Method (MLS) [Liu 2009].

In MLS, each node is associated with a a small region, called influence domain, where its shape function is not equal to zero. The support domain is a set of nodes whose influence domain overlaps the region where the problem weak form is being integrated, which is called integration subdomain. In Figure 1a, for example, the nodes P_1 and P_2 are in the support domain, but the P_3 node is not. On the other hand, in RPIMp the support domain is formed by an arbitrary number of nodes chosen in the integration subdomain neighborhoods (Figure 1b). At first, a box (called level 1, with radius R_1) is taken around the central node, and the nodes within this box are considered to represent the support domain. If there are not enough nodes within this level 1 box, the box is increased (level 2, with radius R_2) to include more nodes. This increment happens until the predetermined number of nodes is reached. The number of nodes within the support domain is a method parameter.

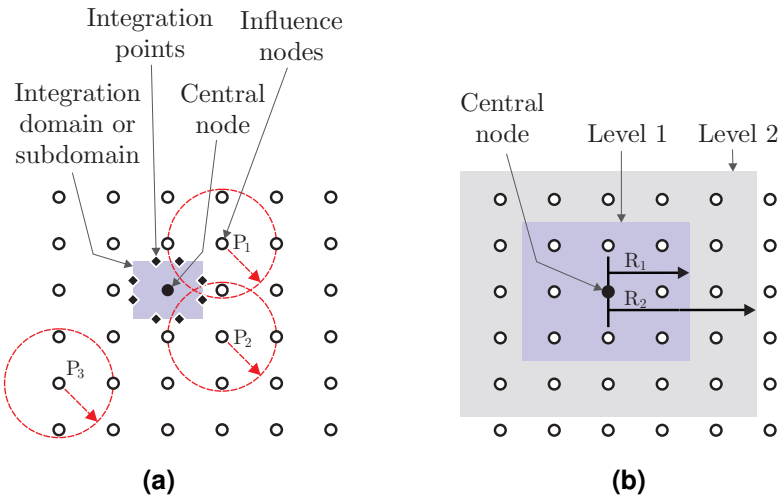

Figure 1. Support domain of MLS (a) and RPIMp (b) [Amorim et al. 2019].

Determining the support domain based on some distance metric is one of the main bottlenecks of meshless methods [Amorim et al. 2019, Ikuno et al. 2013]. One alternative to reduce the computational cost is to explore the parallel nature of the neighborhood location problem, which can be cast to the well-known k-nearest neighbors (kNN) problem, which consists of searching the k closest points to a given point [Garcia et al. 2010]. Insomuch, this work presents a multi-GPU version of the method of locating neighbors based on the space-partitioning data structure, even grid [Amorim et al. 2019, Mei et al. 2016]. The method is adapted so the workload is balanced across multiple GPUs and the integrity of the original method is maintained.

The remainder of this paper is organized as follows. Section 2 formalizes the k-nearest neighbors problem and presents some related works. Section 3 presents the grid method for the kNN problem. Section 4 presents the proposed multi-GPU grid method. Section 5 presents the an overview of CUDA and some implementations details. Section 6 presents the experimental results obtained. Finally, Section 7 presents the conclusions of the present work.

2. K-nearest Neighbors and Related Works

The use of brute force (BF) is one of the most basic approaches for solving kNN problems. In BF methods, the distance between the query point q and every other point in a set of reference points is calculated. Then, the points are sorted by distance and the first k points make up the solution to the problem. Despite the simplicity, BF methods have high computational costs and can become intractable as the number of points increases [Garcia et al. 2010]. Some approaches seek to optimize the BF method by reducing the number of potential neighbors points using spatial partitioning structures, such as R-tree [Kuan and Lewis 1997] and octree [Behley et al. 2015], with good results for low-dimensional spaces.

Recently, General-Purpose Graphic Processing Units (GPGPU) have enabled the use of the graphics processor units to perform previously CPU-only tasks. Due to its high potential for parallelism, it is possible to achieve higher performance than CPUs for many algorithms. Thus, several papers [Garcia et al. 2010, Liang et al. 2009] present parallelized versions of the BF method using single-GPU with satisfactory results. In [Amorim et al. 2019, Mei et al. 2016, Ikuno et al. 2013, Amorim et al. 2020] a single-GPU grid solution is presented. In [Masek et al. 2015], a multi-GPU version of the BF method is presented with good results for low values of k. However, by performing an exhaustive search, the solution becomes impractical for high values of k or for a large number of points.

3. Grid Method

The grid method for the kNN problem consists of partitioning the point space into squares (in two dimensions) or cubes (in three dimensions). Partitioning is implemented to perform only local searches, comparing only points that are real candidates to be neighbors and substantially reducing the number of computed distances. The method consists of three distinct phases: grid creation, point distribution, and local search. For the sake of clarity, we describe the method using a two-dimensional domain. However, a generalization to three-dimensional domains is straightforward.

3.1. Grid creation

The grid is a simple spatial partitioning structure, composed of square cells. For its construction, first the cell width is specified and then the minimum and maximum values of the x and y coordinates of all points are used to determine the planar rectangular region covered by the grid. The number of rows and columns of the grid are easily determined by dividing the rectangle by the width of the cells (Figure 2).

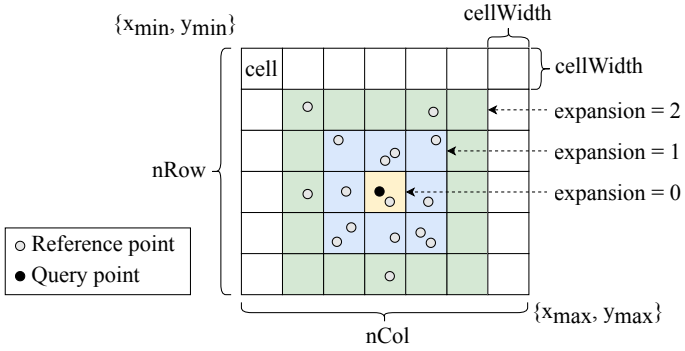

Figure 2. The creation of a planar grid. The white points are part of the set of reference points and the black point is a query point. Three levels of expansion from the query point cell are shown.

3.2. Points distribution

This phase consists of determining the index of the cell to which a given point belongs. For a given point i, the index is given by Equation 1.

$$index_i = \left(\left\lceil \frac{y_i - y_{min}}{cellWidth} \right\rceil \times nCol \right) + \left\lceil \frac{x_i - x_{min}}{cellWidth} \right\rceil \quad (1)$$

To improve data locality, the points are sorted by their cell index so that all points from the same cell are stored contiguously in the memory. Thus, it is only necessary to store the number of points in each cell and the index of the first point of each cell to find all points that belong to a given cell in constant time.

For the subsequent implementation of the local search, an expansion factor for each cell is pre-calculated. This factor is responsible for delimiting a local search region to all points associated with the same cell and for ensuring that all the closest neighbors of the points in this cell are present in this region. The process for determining it consists of expanding one level from a given cell until the number of points in the formed region is greater than or equal to k. When no further expansions are needed, Equation 2 is applied to the value found to ensure that the search for neighbors is always exact.

$$expansion = \left\lceil \sqrt{2} \times (expansion + 1) \right\rceil \quad (2)$$

Figure 2 shows three levels of expansion of a cell. In this way, it is guaranteed that all k-nearest neighbors of the points of a given cell are in the region delimited by its expansion factor.

Figure 3 illustrates the organization of all information needed to represent the grid and to perform local searches.

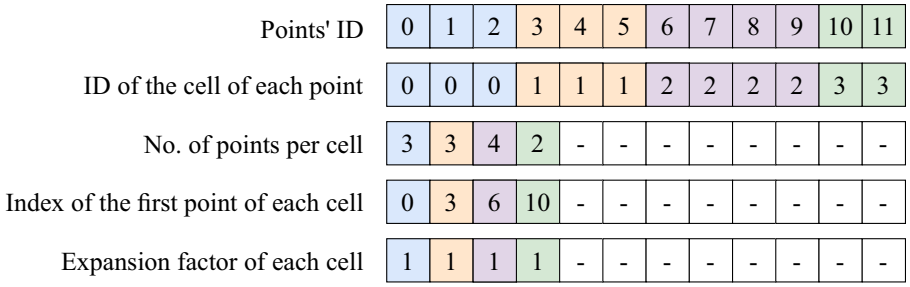

Figure 3. Organization of the necessary information to represent the grid and to carry out local searches.

3.3. Local search

The local search consists of calculating the distance between a query point and all reference points in the region delimited by the expansion factor of the cell to which the query point belongs and selecting the k shortest distances. This process is repeated for all query points and can be summarized in the three following steps:

1. Find the cell where the query point is located (Equation 1);
2. Find the region delimited by the expansion factor of that cell (Figure 2);
3. Calculate the distance between the query point and all points in this region and select the k shortest distances.

4. Multi-GPU Grid Method

This section presents our proposed grid method adaptation for multi-GPU. The key idea is to distribute and organize the reference points between the GPUs so that each query needs to be executed in only one GPU and that the amount of duplicate reference points between the GPUs is minimal. For the sake of clarity, the multi-GPU grid method is described for two GPUs and also with a two-dimensional domain.

4.1. Indexing phase

We consider that the set of reference points is initially stored in the CPU memory and does not have any kind of sorting. The indexing phase begins by splitting equally and randomly the reference points between the GPUs. Each GPU creates its own grid, as presented in Section 3.1. However, all grids are created with the same dimensions, that is, the minimum and maximum coordinates of all points are considered and not only of the points assigned to each GPU. Then, each GPU distributes the points assigned to it on its grid (as shown in Section 3.2).

To ensure that each query needs to be run on only one GPU, the reference points are rearranged between the GPUs. A grid row (row_m) that splits the grid into two halves is determined in such way that the amount of points in each half is approximately equal. Then each GPU will store one half of the grid and be responsible for running its queries.

To determine the row_m, a reduction operation is performed on the points per cell vector of both GPUs. In this way, it is possible to determine, globally, the number of

points on each grid row and then find the best row of division. Then, the reference points previously assigned to GPU_1 that are below the row_m are transferred to GPU_2 and the reference points previously assigned to GPU_2 that are above the row_m are transferred to GPU_1.

The split presented above does not take into account that query points located near the row_m may have neighbors that have been assigned to another GPU. Thus, it is necessary that the reference points of a small region near row_m be duplicated in both GPUs. For GPU_1, this region is determined by finding the furthest row (row_f) from the area assigned to GPU_2 that the expansion factor of any of its cells has reached. Therefore, all GPU_2 points between the row_m and the row_f must also be present in the GPU_1. Similarly, the process is carried out for GPU_2.

Figure 4 illustrates the steps of the indexing phase as well as the search phase. Note that the operations presented in Section 3.2 are performed in a different order to avoid unnecessary transfer of information among GPUs.

4.2. Search phase

As shown above, the reference points indexing guarantees that all the neighbors of a query point will be present on the same GPU. Thus, the search phase consists of determining the cell in which each query point is located and transferring it to the GPU responsible for that cell. Once in the correct GPU, the local search is performed as shown in Section 3.3.

5. CUDA Implementation Details

CUDA is a general-purpose parallel computing platform and computing model developed by NVIDIA in 2006 [NVIDIA 2020], and it is the programming model used in this work.

In this programming model, the functions executed in parallel are called kernels. When called by the host CPU, a kernel is executed N times in parallel by N threads on the GPU. The threads are organized in a hierarchy of blocks and grid of blocks. Blocks are sets of threads that run on the same processor core and share a private memory reserved for the block. The grid, in turn, is a set of blocks and defines the total set of threads created and managed by the GPU at the kernel launch [NVIDIA 2020].

The critical parts of the grid creation and of the points distribution were carried out with the help of the Thrust library, which is a well-known library of parallel algorithms and data structures. For instance, the function `thrust::sort_by_key` was used to sort the points by the cell index and the functions `thrust::reduce_by_key` and `thrust::unique_by_key` were used to determine the number of points per cell and the index of the first point of each cell, respectively. To perform the search, a kernel was created and each thread was responsible for processing a query point. As the query points are sorted by the index of the cells, each thread warp will execute queries that are spatially close and, therefore, it is possible to perform coalesced memory access and minimize the divergence of the execution path of each thread. Data transfers and operations between GPUs were handled by the NCCL library, which provides optimized collective communication primitives tailored for NVIDIA GPUs.

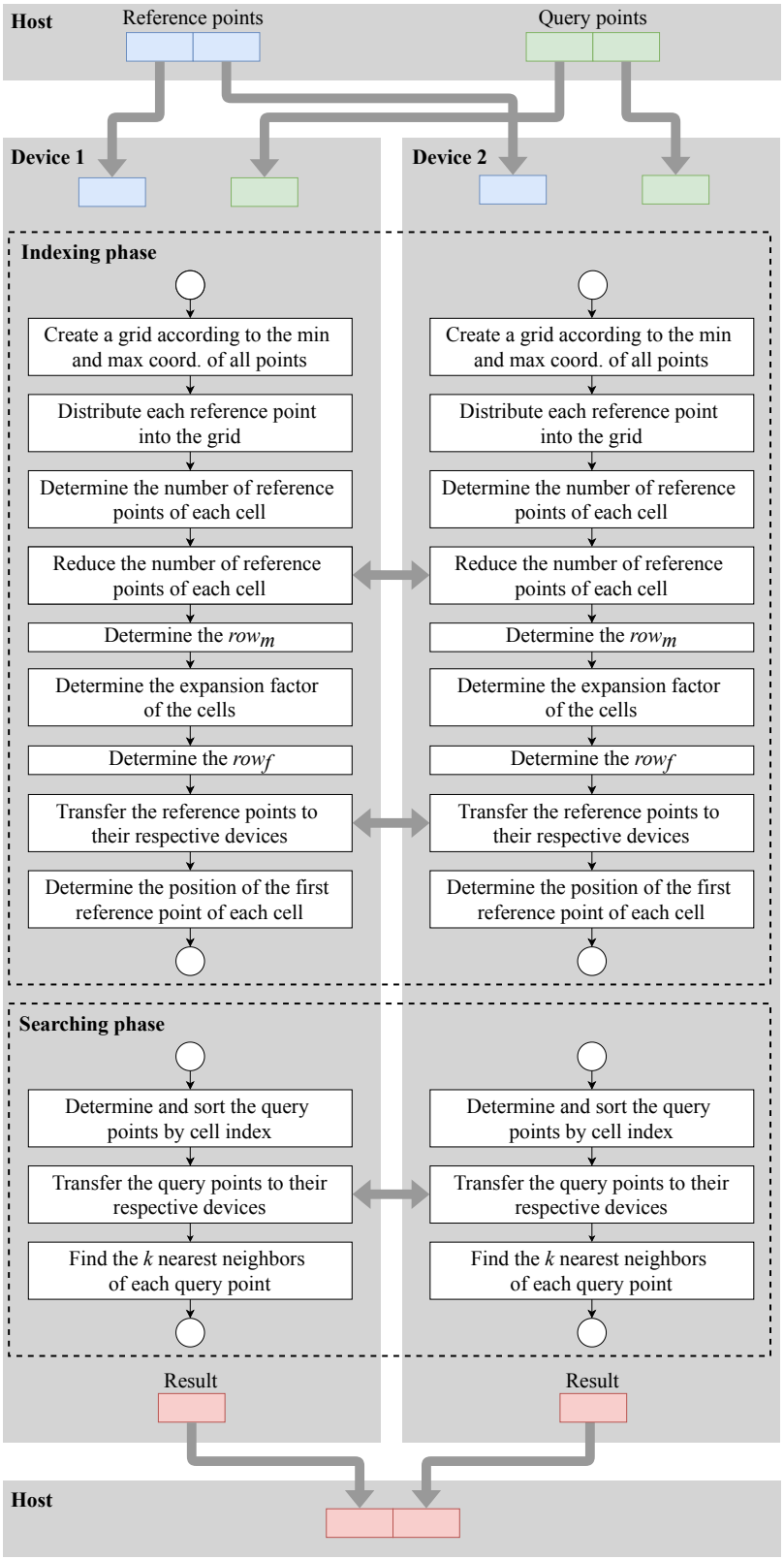

Figure 4. Overview of the indexing and search phases using two GPUs (Device 1 and Device 2). The sets of reference and query points start in the CPU (host) memory and the final result is also given in the CPU memory. The thick gray arrows represent data transfer between host/device or device/device and the black arrows represent the flow of the algorithm.

6. Experimental Results

All tests were performed in a development environment containing 2 × Intel Xeon CPU E5-2620 v2 2.10 GHz with 64 GB RAM DDR3 1600 MHz and 4 × GPU NVIDIA Tesla K40m with 12GB GDDR5 SDRAM.

First, the execution time of the single-GPU grid method was compared with both the CPU and single-GPU version of the brute-force method as a baseline performance comparison (Section 6.1). Next, we compared the execution time of the single-GPU grid method with the proposed multi-GPU grid method using two and four GPUs (Section 6.2).

The sets of points used in all tests are two-dimensional and were generated randomly. The coordinates of each point are of the float type and range from 0 to 10^5. In each test case, the set of query points is equal to the set of reference points, i.e., the number of query points increases proportionally with the number of reference points.

6.1. Single-GPU Grid vs. CPU BF vs. Single-GPU BF

Despite its well-known poor performance for large datasets, the brute-force method for kNN search is still widely used in many applications that needs to perform search on small datasets.

The FAISS library [Johnson et al. 2019], developed and maintained by the Facebook AI Research group, provides an optimized implementation of the brute-force method, for both CPU and GPU, and is therefore used as a base line performance comparison for our grid method implementation.

The Figure 5 shows the processing time of the CPU brute-force method (from FAISS), single-GPU brute-force method (from FAISS) and single-GPU grid method (our implementation) with a fixed value of k (100) and varying the number of points (from 10^4 to 10^6). It is possible to notice, as expected, that the GPUs methods are orders of magnitude faster than the CPU one.

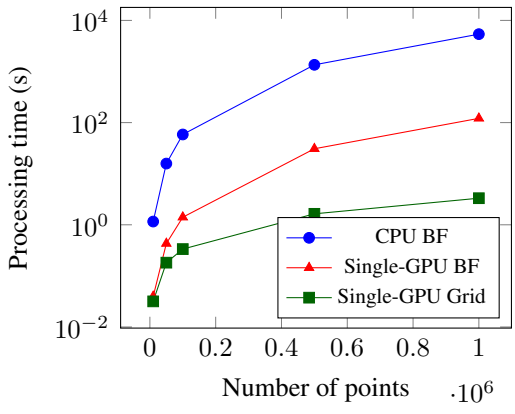

Figure 5. kNN execution time using CPU brute-force method, single-GPU brute-force method and single-GPU grid method with fixed value of *k* (100) and varying the number of points.

The Figure 6, in turn, shows the speedup achieved using the single-GPU grid method against the CPU and single-GPU brute-force method. The single-GPU grid method proved to be more than $1600\times$ faster than the CPU brute-force method with a number of points equal to 10^6 and k equal to 100. Against the single-GPU brute-force method, it was possible to achieve a speedup of more than $36\times$ with that same amount of points and k value.

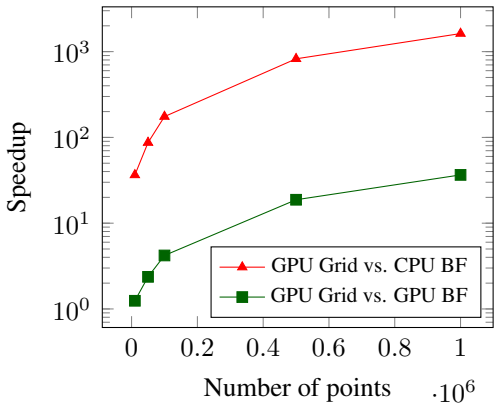

Figure 6. Speedup achieved using single-GPU grid method against CPU brute-force method and single-GPU brute-force method with fixed value of *k* (100) and varying the number of points.

6.2. Single-GPU Grid vs. Multi-GPU Grid

Finally, the proposed multi-GPU grid method was compared with the single-GPU grid method. Figure 7 shows the processing time of the grid method using one, two and four GPUs with a fixed value of k and varying the number of points (Figure 7a) and the processing time with a fixed number of points and varying the value of k (Figure 7b). It is possible to infer that for low values of k or for low amounts of points, the multi-GPUs approach is not recommended due communication and synchronization overheads.

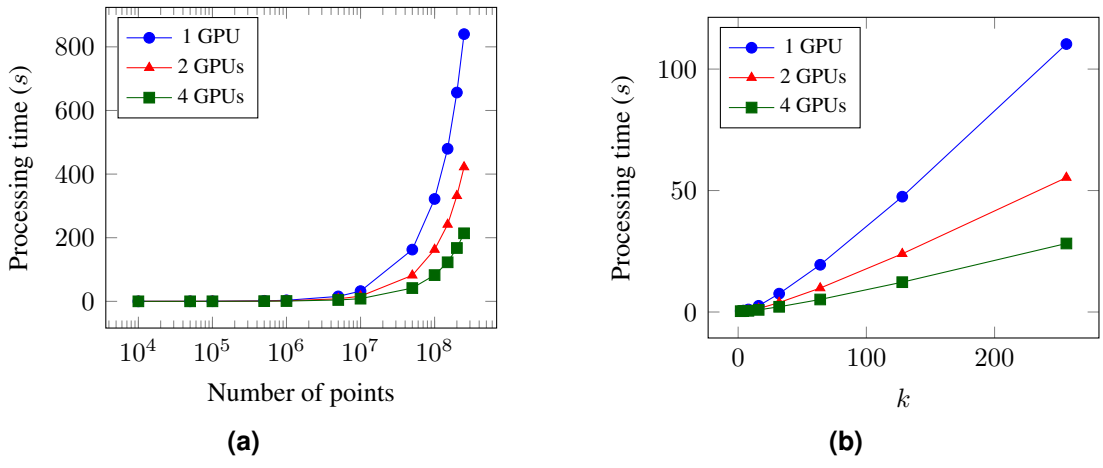

Figure 7. kNN execution time using the single and multi-GPU grid method: fixed value of *k* (100) and varying the number of points (Figure 7a); fixed number of points (10^7) and varying the value of *k* (Figure 7b).

However, as shown in Figure 8, it is possible to achieve an almost linear speedup when k or the number of points is sufficiently high. For instance, with 2.5×10^8 points and k equal to 100, it was possible to achieve the speedup of $1.99\times$ and $3.92\times$ with two and four GPUs, respectively, against single-GPU. With the number of points equal to 10^7 and k equal to 256, it was possible to reach the speedups $1.99\times$ and $3.94\times$ using two and four GPUs, respectively, against a single-GPU.

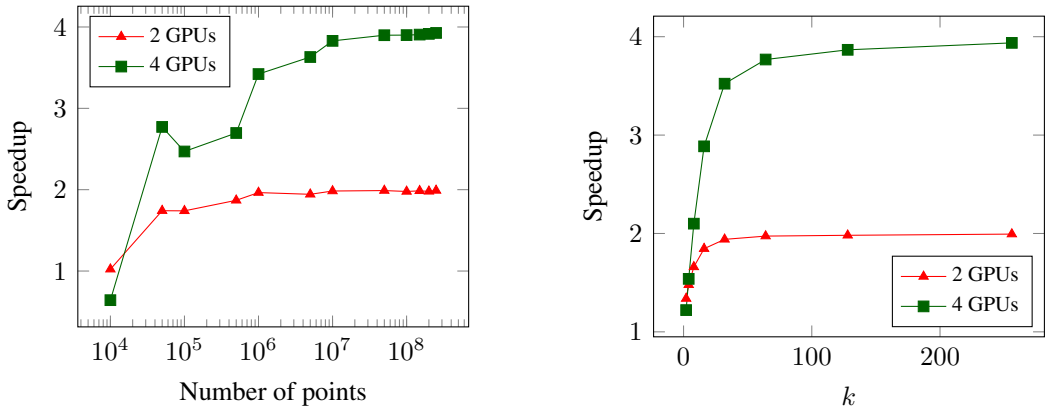

Figure 8. Speedup achieved using multi-GPU against single-GPU: fixed value of *k* (100) and varying the number of points (Figure 8a); fixed number of points (10^7) and varying the value of *k* (Figure 8b).

7. Conclusions

One of the main bottlenecks in the processing of meshless methods is determining the neighborhood information between the nodes, task which can be cast to the kNN problem. Thus, this work proposed a multi-GPU version of the grid method for solving the kNN problem. Compared to the single-GPU version, it was possible to achieve a speedup of up to $1.99\times$ using two GPUs and up to $3.94\times$ using four GPUs, when the number of nodes or the value of k were sufficiently high to fully employ the GPUs resources.

References

Amorim, L., Goveia, T., Mesquita, R., and Baratta, I. (2020). GPU Finite Element Method Computation Strategy Without Mesh Coloring. *Journal of Microwaves, Optoelectronics and Electromagnetic Applications*, 19:252 – 264.

Amorim, L. P., Mesquita, R. C., Goveia, T. D. S., and Correa, B. C. (2019). Node-to-Node Realization of Meshless Local Petrov Galerkin (MLPG) Fully in GPU. *IEEE Access*, 7:151539–151557.

Behley, J., Steinhage, V., and Cremers, A. B. (2015). Efficient Radius Neighbor Search in Three-dimensional Point Clouds. In *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2015-June, pages 3625–3630. Institute of Electrical and Electronics Engineers Inc.

Chen, J. S., Hillman, M., and Chi, S. W. (2017). Meshfree Methods: Progress Made after 20 Years. *Journal of Engineering Mechanics*, 143(4):04017001.

Du, Q., Wang, D., and Zhu, L. (2009). On Mesh Geometry and Stiffness Matrix Conditioning for General Finite Element Spaces. *SIAM Journal on Numerical Analysis*, 47(2):1421–1444.

Garcia, V., Debreuve, E., Nielsen, F., and Barlaud, M. (2010). K-nearest Neighbor Search: Fast GPU-based Implementations and Application to High-dimensional Feature Matching. In *ICIP Proceedings*, pages 3757–3760.

Garg, R., Chandra Thakur, H., and Tripathi, B. (2015). A Review of Applications of Meshfree Methods in the area of Heat Transfer and Fluid Flow: MLPG method in particular. *International Research Journal of Engineering and Technology*, 1(2007):329–338.

Geuzaine, C. and Remacle, J.-F. (2009). GMSH: A 3-D Finite Element Mesh Generator With Built-in Pre and Post-processing Facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331.

Ikuno, S., Fujita, Y., Hirokawa, Y., Itoh, T., Nakata, S., and Kamitani, A. (2013). Large-Scale Simulation of Electromagnetic Wave Propagation Using Meshless Time Domain Method With Parallel Processing. *IEEE Transactions on Magnetics*, 49(5):1613–1616.

Jin, J.-M. (2015). *The Finite Element Method in Electromagnetics*. John Wiley & Sons.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale Similarity Search With GPUs. *IEEE Transactions on Big Data*, pages 1–1.

Kamranian, M., Dehghan, M., and Tatari, M. (2017). An Adaptive Meshless Local Petrov–Galerkin Method Based on a Posteriori Error Estimation for the Boundary Layer Problems. *Applied Numerical Mathematics*, 111:181–196.

Kuan, J. and Lewis, P. (1997). Fast k Nearest Neighbour Search for R-tree Family. In *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat. No.97TH8237)*, pages 924–928. IEEE.

Liang, S., Wang, C., Liu, Y., and Jian, L. (2009). CUKNN: A Parallel Implementation of K-nearest Neighbor on CUDA-enabled GPU. In *YC-ICT2009 Proceedings*, pages 415–418.

Liu, G. R. (2002). *Element Free Methods*. CRC, 1 edition.

Liu, G. R. (2009). *Meshfree Methods: Moving Beyond the Finite Element Method*. CRC Press, 2 edition.

Liu, G. R. (2016). An Overview on Meshfree Methods: For Computational Solid Mechanics. *International Journal of Computational Methods*, 13(05):1630001.

Masek, J., Burget, R., Karasek, J., Uher, V., and Dutta, M. K. (2015). Multi-GPU Implementation of k-nearest Neighbor Algorithm. In *TSP2015 Proceedings*, pages 764–767. Institute of Electrical and Electronics Engineers Inc.

Mei, G., Xu, N., and Xu, L. (2016). Improving GPU-accelerated Adaptive IDW Interpolation Algorithm Using Fast kNN Search. *SpringerPlus*, 5:1–22.

NVIDIA (2020). *CUDA C++ PROGRAMMING GUIDE - Design Guide*. NVIDIA.