Parallel Structured Mesh Generation with Disparity Maps by GPU Implementation

Hongjian Wang, Naiyu Zhang, Jean-Charles Créput, Julien Moreau and Yassine Ruichek

Abstract—The goal of structured mesh is to generate a compressed representation of the 3D surface, where near objects are provided with more details than objects far from the camera, according to the disparity map. The solution is based on the Kohonens Self-Organizing Map algorithm for the benefits of its ability to generate a topological map according to a probability distribution and its potential to be a natural massive parallel algorithm. The disparity map, which stands for a density distribution that reflects the proximity of objects to the camera, is partitioned into an appropriate number of cell units, in such a way that each cell is associated to a processing unit and responsible of a certain area of the plane. The advantage of the proposed model is that it is decentralized and based on data decomposition. The required processing units and memory are with linearly increasing relationship to the problem size. Experimental results show that our GPU implementation is able to provide near real-time performance with small size disparity maps and the running time increases in a linear way with a very weak increasing coefficient. The proposed method is suitable to deal with large scale problems in a massively parallel way.

Index Terms—Parallel Cellular Model, Mesh Generation, Disparity Map, Self-Organizing Map, GPU Implementation

1 Introduction

In the field of artificial vision, robot navigation and 3D surface reconstruction, a lot of work has been done to develop effective stereo matching algorithms producing high quality disparity maps. Such a disparity map is obtained by a matching process of the two left and right images obtained by a stereo camera. But few approaches of stereo matching and surface reconstruction currently match the requirements of real-time execution. The volume of data stored and manipulated in the disparity map often constitute an obstacle for fast algorithms.

Parallel algorithms on graphics processing units (GPU) are now developed to respond to this problematic issue. Here, we address the problem of building, by parallel GPU implementation, a compressed and adapted structured mesh representing a given disparity map. The goal is to manifest the 3D scene in an efficient way, with more details for objects close to the camera and less details for objects far from it. The compressed structured mesh obtained in 2D space, as for the disparity map, should then allows fast treatment or visualization in 3D space. To achieve such a goal, we choose to implement the Kohonen's self-organizing map (SOM) algorithm [1], [2] in a massively parallel way on GPU. (Here we use "massive parallelism" in a sense that will be clarified later in this very section.) The disparity map is the input. It is used as a 2D density distribution, on which the algorithm operates by deformation of a 2D hexagonal grid, called the neural network. The SOM can be seen as a center based clustering algorithm with topological relationships between cluster centers. The neural network is a visual pattern that adapts and modifies its shape according to some underlying distribution.

The size of the grid is lower than and in relation to the size of the disparity map in such a way that the neural network constitutes a compressed representation of the disparity map, with a parameterized compression rate. The grid deformation must respect the density distribution and topology. This means that high disparity values, which correspond to objects close to the camera, are represented by higher densities of neural network points and that the structured network reflects the spatial topology or distances in 2D and 3D space. Proximity of grid points reflects proximity in Euclidean space. The surface reconstruction in 3D space obtained by using the adapted mesh can be seen as a compressed representation of the 3D surface, such that objects close to the camera have higher resolution and their details are more finely represented.

Unlike most parallel models of metaheuristic optimization, which are either based on data duplication or mixed sequential/parallel computation, one important feature of the parallel model presented in this paper, is that it proceeds from a cellular decomposition of the input data, i.e. the disparity map, in 2D space, such that each processing unit represents a constant and small part of data. Hence, as more and more multi-cores will be available in a single chip (instead of only 15 streaming multi-processors in our GPU

• *The authors are with the Systems and Transportation Laboratory (SeT) of the Research Institute on Transportation, Energy and Society (IRTES), University of Technology of Belfort-Montbéliard (UTBM), Belfort, 90000, France.*
E-mail: bennyboy.shu@gmail.com, hongjian.wang@utbm.fr

platform) in the future, the approach should be more and more competitive, especially when dealing with very large size inputs. This is because in our approach the required processing units and memory are with linearly increasing relationship to the problem size. This property is what we call "massive parallelism".

The rest of this paper is organized as follows. Section 2 describes the general mesh generation problem, and then focuses on the definition of balanceable structured mesh. Section 3 highlights the principle of applying SOM algorithm to structured mesh problem. In Section 4, elaborate design of parallel cellular model is illustrated. A thorough GPU implementation with considerations of parallelism control and memory management is provided in Section 5. Section 6 reports experiments including analyses of performance influencing factors and results obtained from inputs with different sizes. Finally, some conclusions of this work are drawn in Section 7.

2 Problem Statement

2.1 Mesh Generation

A structured mesh in the plane is generally a grid of vertices deformed by some coordinate transformation. Each vertex of the mesh, except at the boundaries, has an isomorphic local neighborhood. Structured meshes correspond to the three possible tessellations of the plane with identical regular polygons: square, triangular, and hexagonal, whereas an unstructured mesh is most often a triangulation with arbitrarily varying local neighborhoods. Structured or unstructured mesh generation can be used in many fields to improve the precision of simulation-based computations by optimizing the choice of points to be considered in the simulation, relatively to the underlying resource.

Most methods for surface reconstruction generally deal with Delaunay triangulation and/or recursive subdivision that yield to unstructured meshes with variable neighborhoods at different level of refinement, and hence these methods require specific and complex data structures. The generation of structured meshes according to a density distribution is another meshing technique useful for surface reconstruction. In this case, the advantage can reside in the grid data structure with fixed dimensions, and constant neighborhood that allows fast access computation and compact representation.

While mesh generation is a large area of research [3]–[9], few approaches exist for structured mesh generation according to an underlying density distribution, in order to represent data by density and topology preservation. For instance, the approaches described in [10]–[12] deal with this subject but the adaptive mesh only carries out a rough approximation according to the underlying traffic map. Also, their implementations are sequential. Authors of [13], [14] have proposed fast generation methods for meshes but as their methods are too memory consuming, they can not handle large size problems. As a main hypothesis it results from these related works, that the structured mesh generation is a new problem; and it is an NP-hard problem related to usual balanced clustering problems in the plane [15], [16]. It looks like that GPU parallel computation is still an open field for such problems and applications. To the best of our knowledge, we did not find any parallel implementations for structured mesh applications based on cellular decomposition of the input data, like we propose in this paper.

2.2 Balanceable Structured Mesh

In this paper, the application to structured mesh generation is presented as the solution of a balanced clustering optimization problem in the plane. The goal is to homogeneously divide a density distribution, the disparity map in our application, between many triangles of the structured hexagonal mesh.

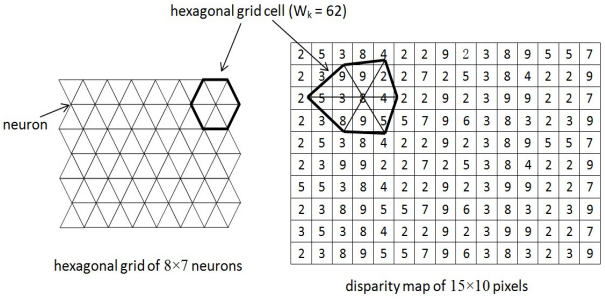
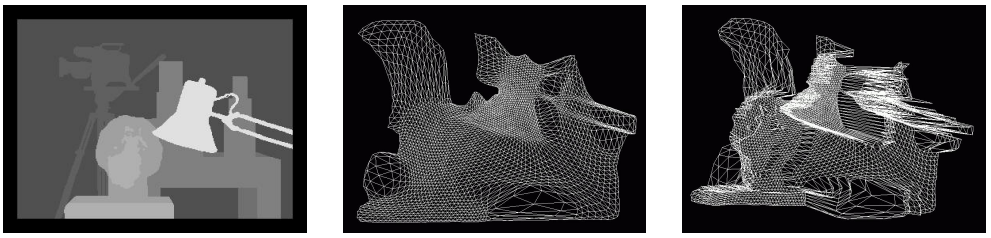


Fig. 1: Hexagonal structured mesh with honeycomb cells and triangular subdivision (left part). Disparity map covered with a single honeycomb cell with weight $W_k = 62$ (right part).

The definition of balanceable structured mesh optimization problem needs to consider both the hexagonal grid (the mesh) and the underlying density distribution (the disparity map). The target adaptive hexagonal mesh is illustrated in the left part of Fig. 1. It is defined as a set of hexagonal cells, each one containing six subdivided triangles. These basic honeycomb cells are the units used to evaluate the amount of the underlying pixels they cover in the disparity map. The right part of Fig. 1 shows such a hexagonal cell and its covering pixels from the disparity map. In this example, the total value covered, called the weight of the honeycomb cell, is the summation of the underlying pixel values ($W_k = 62$).

Let M_k be the set of honeycomb cells of the mesh. Let W_k be the weight of a single honeycomb cell. Note that this weight can be computed by using a standard pixel coloring algorithm. Let W be the average weight of the K honeycomb grid cells defined by Equation 1. We define the optimization problem as the minimization of the average percentage deviation of each individual honeycomb cell weight to the average honeycomb cell weight as defined in Equation



(a) (b) (c)

Fig. 2: A meshing example: (a) Input disparity map. (b) Meshing result viewed in 2D space. (c) Meshing result viewed in 3D space.

2. Hence, the structured mesh generation problem consists in minimizing this criteria while preserving regularity of hexagonal topology, such geometrical constraints being only visually verified in this paper.

$$W = \frac{\sum_k W_k}{K} \tag{1}$$

$$\%\ cost = \frac{\sum_k |W_k - W|}{K \times W} \times 100 \tag{2}$$

Fig. 2 demonstrates a structured mesh example by our application. The *tsukuba* disparity map from the Middlebury Stereo Datasets [17] is the input as shown in Fig. 2(a). Fig. 2(b) and (c) are the meshing result viewed form 2D space and 3D space respectively. In the disparity map, brighter regions are nearer to the camera view point, such as the lamp; and in the meshing result, the adapted grid presents higher density on such regions, with respect of the topology of the scene.

3 SOM FOR STRUCTURED MESH

The Kohonen's self-organizing map (SOM) [1], [2] algorithm is a neural network approach dealing with visual patterns moving and adapting themselves to brute distributed data in space. It is often presented as a non supervised learning procedure performing a non parametric regression that reflects topological information of the input data. The standard SOM is a non directed graph $G = (V, E)$, called neural network, or topological grid, or structured mesh, where each vertex $v \in V$ is a neuron having a synaptic weight vector $w_v = (x, y) \in \Re^2$. Here, $\Re^2$ is the two-dimensional Euclidean space. Synaptic weight vector corresponds to the vertex location in the plane. The set of neurons V is provided with the d_G induced canonical metric $d_G(v, v') = 1$ if and only if $(v, v') \in E$, and with the usual Euclidean distance $d(v, v')$.

When applied to a structured mesh with hexagonal topology, the training procedure consists of a fixed amount of t_{max} iterations that are applied to a mesh, with the vertex coordinates of the mesh being initialized to a regular grid (each vertex has 6 neighbors). Here, data set is the set of pixels of the disparity map. Note that the disparity map stands for a density distribution, where each pixel value represents a density value. Each iteration follows three basic steps. At each iteration t, a point $p(t) \in \Re^2$ is randomly extracted from the data set (extraction step) according to a roulette wheel mechanism depending on the disparity values. Then, a competition between neurons against the input point *p(t)* is performed to select the winner neuron n^* (competition step). Usually, it is the nearest neuron to *p(t)*. Finally, the learning law (triggering step) presented in Equation 3 is applied to n^* and to the neurons within a finite neighborhood of n^* with radius σ_t, in the sense of the topological distance d_G, using learning rate $\alpha(t)$ and function profile h_t. The function profile is given by a Gaussian form in Equation 4. Here, the learning rate $\alpha(t)$ and radius σ_t are geometric decreasing functions of time. To perform a decreasing run within t_{max} iterations, at each iteration *t*, the coefficients $\alpha(t)$ and σ_t are multiplied by $exp(ln(\chi_{final}/\chi_{init})/t_{max})$ with respectively $\chi = \alpha$ and $\chi = \sigma$, χ_{init} and χ_{final} being respectively the values at the starting and the final iteration.

$$w_n(t+1) = w_n(t) + \alpha(t) \times h_t(n^*, n) \times (p(t) - w_n(t)) \tag{3}$$

$$h_t(n^*, n) = exp(-\frac{d_G^2(n^*, n)}{\sigma_t^2}) \tag{4}$$

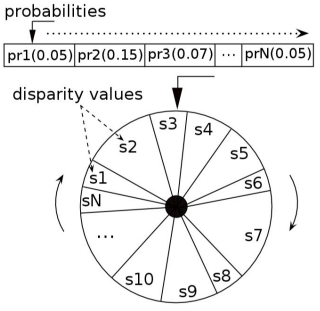


Fig. 3: Roulette wheel selection. Each pixel in the disparity map gets a portion of the wheel according to its disparity value.

The roulette wheel mechanism in the extraction step is depicted as in Fig. 3. For each pixel (point *p*) of the input disparity map, its probability (*pr*) to be chosen is defined by Equation 5, where N is the total number

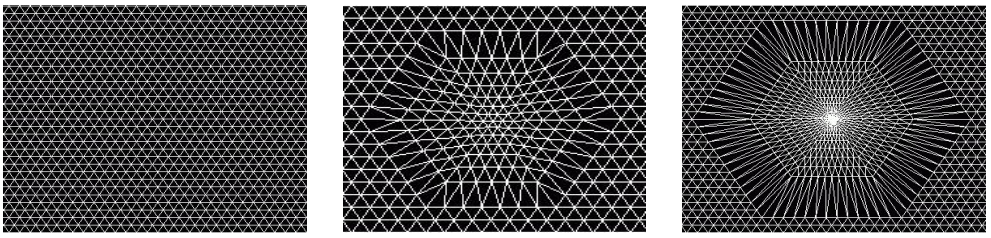


Fig. 4: A single SOM iteration with learning rate α and radius σ. Left: Initial configuration. Middle: $\alpha = 0.5, \sigma = 6$. Right: $\alpha = 1, \sigma = 12$.

of pixels in the disparity map and s is the pixel's disparity value.

$$pr_i = \frac{s_i}{\sum_{j=0}^{N} s_j} \tag{5}$$

Examples of a basic iteration with different learning rates and neighborhood sizes are shown in Fig. 4. Note that a SOM simulation is characterized by the five running parameters: $\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}, t_{max}$. Application of SOM to the stereo disparity map consists of applying the training procedure to a structured mesh (the neural network) according to a density distribution that is a simple transformation from the disparity map. This transformation consists, before the training starts, in removing background values (very low disparity values) and increasing contrast of disparity values. This is done in order to increase the data point density for closest objects in the image. Here, density values are set to the cube of the disparity values.

Considering SOM applications on structured meshing problems, few works refer to parallel execution or real-time requirement. According to [18]–[20], researchers integrate SOM with meshes and surface reconstructions, but never refer to any space decomposition for further studies of parallel implementations. [21] proposes a simple combination of SOM and cellular automata in meshing processing, but the paper foregoes further exploration either in the properties of this combination or the possibility of execution on parallel architectures. In [22], the authors present a SOM based algorithm for implicit surface reconstruction. However, neither parallel implementation is provided nor real-time requirement is concerned. To the best of our knowledge, we did not find any GPU parallel implementation of the SOM algorithm when applied in 2D or 3D space for structured mesh applications. We also did not find any cellular data/space decomposition-based method for parallelism. With regard to computing SOM on GPU, some methods have been proposed in [23], [24]. These methods accelerate SOM process by parallelizing the inner steps in each basic iteration, and they mainly focus on two aspects as follows, firstly, to find out the winner neuron in parallel, secondly, to move the winner neuron and its neighbors in parallel. In our model, we use each parallel processing unit to perform SOM iterations independently in parallel, each one to a part of the data, instead of using many parallel processing units to cooperatively accelerate some part of a sequential SOM procedure iteration by iteration. The proposed SOM model is aimed at implementing decentralized distributed computation, which does not depend on some central control. This parallelizing strategy makes our approach robust and scalable, since processing units increase the same way as the data size.

4 Parallel Cellular Model Design

4.1 Cell Partition

It is intuitive that the disparity map and the structured mesh built by the SOM algorithm are connected by sharing the same Euclidean plane. The plane is defined by the two-dimensional disparity map, and each neuron can move on the disparity map/plane. Given an input disparity map with size of $W \times H$, a two-dimensional topological grid/neural network will be created, with size of $W/g \times H/g$. Note that the size of the neural network is in relation to the size of the disparity map in such a way that the neural network constitutes a compressed representation of it. The compression rate is the quotient of the sizes g^2. Note also that each neuron has coordinates (weights) in the Euclidean plane, and these coordinates are defined by the dimensions of the density map.

The standard online SOM algorithm is sequential owing to its iterative training process. In order to implement the parallel level, on which parallel execution will take place, we introduce a supplementary level of decomposition of the plane and input data. Between the topological grid/neural network and the disparity map, we add a uniform two-dimensional cellular matrix with size $W/co \times H/co$, where co is a constant factor to control the degree of parallelism. The three main data structures of the parallel model are illustrated in Fig. 5. This intermediate cellular matrix is in linear relationship to the input size. Its role is to memorize the neurons in a distributed fashion and authorize many parallel closest point searches in the plane by a spiral search algorithm [25], and many parallel training procedures.

Each uniformly sized cell in the cellular matrix is a basic training unit and will be handled by one

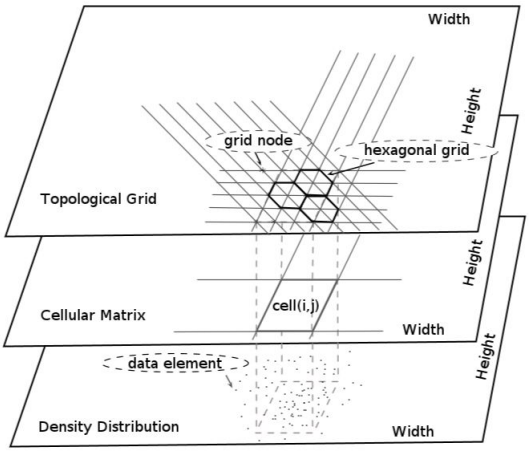


Fig. 5: Parallel cellular model: the input density distribution, the cellular matrix and the topological grid. To a given cell of the cellular matrix corresponds a part of the input data as well as a part of the structured hexagonal mesh.

parallel processing unit, here a thread in our GPU implementation, during the iterations of the SOM processing. This is the level on which massive parallelism takes place. Since the cellular matrix division is proportional to the input disparity map, and the processing units correspond one-to-one with the cells respectively, then, the processing units are also in linear correspondence to the input disparity map size, in $O(W \times H)$.

Each processing unit will have to perform the different steps of the sequential SOM iteration in parallel. A problem that arises is then to allow many data points extracted at the first step by the processing units, at a given parallel iteration, to reflect the input data density distribution. As a solution to this problem, we propose a particular cell activation formula stated in Equation 6 to choose cells that will execute or not at the considered iteration.

$$pra_i = \frac{S_i}{\max\{S_1, S_2, \ldots, S_{num}\}} \times \delta \qquad (6)$$

Here, pra_i is the probability that the cell i will be activated, S_i is the sum of disparity values of all the pixels in the cell i, and num is the number of cells. The empirical preset parameter δ is used to adjust the degree of activity of cells/processing units, in order to avoid too many memory access conflicts. As a result, the higher disparity value a cell contains, which means the denser this cell is, the higher is the probability this cell to be activated to carry out the SOM execution at each parallel iteration. In this way, the cell activation depends on a random choice based on the input data density distribution.

4.2 Cellular-Based Parallel SOM

Based on the cell partition, we have to define parallel SOM processes that are implemented by many cells/processing units corresponding to the cellular matrix decomposition. The goal is to carry out five steps on the basis of SOM algorithm in a distributed way. These steps are: (1) cell activation step, (2) input data point extraction step according to the disparity map, (3) closest point, or winner neuron, finding step, (4) application of learning rule step, (5) parameters decrease step. Then, this process, performed independently in parallel by many cells, will be repeated a given number of times as stated by the parameter t_{max} of the original SOM algorithm. Note that t_{max} is now the number of parallel executions, rather than the number of sequential iterations.

A cell is activated or not depending on the activation probability. If the cell is activated, the processing unit will continue to perform the next three parallel operations, otherwise it does nothing and directly skips to the end of the current iteration.

In the parallel extraction step, each activated cell performs a local roulette wheel mechanism in the cell itself, in order to get the extracted pixel. The probability of a pixel choice local to a cell is defined by Equation 7, where *Npic* is the number of pixels in the cell, and s_i is the disparity value of a pixel in the cell.

$$pr'_i = \frac{s_i}{\sum_{j=0}^{Npic} s_j} \qquad (7)$$

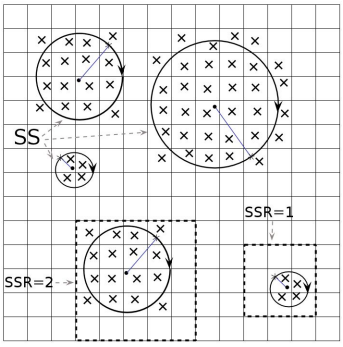


Fig. 6: Demonstration of spiral searches (SS) performed in parallel and spiral search with range (SSR).

In the competition step, the processing unit carries out a spiral search [25] based on the cell partition model to find the nearest neuron to the extracted point. The cell in which this point lies will be searched first. If this cell is empty of neurons, then the cells surrounding it are searched one by one in a spiral-like pattern until a neuron is found. Once a neuron is found, it is guaranteed that only the cells that intersect a particular circle, which is centered at the extracted point and with the radius equal to the distance between the first found neuron and the extracted point, have to be checked before finishing searching, as the cells with "X" marks shown in Fig. 6. It is worth noting that a single spiral search process takes *O(1)* computation time on average for a bounded distribution according to the instance size

[25]. Then, one of the main interests of the proposed approach is to allow the execution of approximately N spiral searches in parallel, where $N = W \times H$ is the problem size, and thus transforming a $O(N)$ sequential algorithm search into a parallel algorithm with theoretical constant time $O(1)$ in the average case for bounded distributions. This is what we call "massive parallelism", the theoretical possibility to reduce computation time by factor N, when solving a Euclidean NP-hard optimization problem.

In the triggering step of learning, each processing unit performs the displacements of the neurons in the plane starting from the closest one (winner neuron), which is found in the competition step. The movements of relevant neurons follow the learning rule of Equation 3 and Equation 4 discussed in Section 3 and they involve not only the closest neuron but also its neighbors in the sense of topological distance into a neighborhood with radius σ_t , as shown in Fig. 4. All the processing units share one unique neural network in the Euclidean space. The coordinates of neurons are therefore stored into a shared buffer which is simultaneously accessed by all the parallel processing units.

The last step is to decrease the learning parameters. After all the parallel processing units have finished their jobs in one single iteration, the learning rate α and radius σ are decreased, getting ready for the next parallel iteration. For t_{max} parallel iterations, the maximum number of single SOM iterations is $t_{max} \times (W/co \times H/co)$, which corresponds to the extreme case where all the processing units are activated at the same time.

5 GPU IMPLEMENTATION

We use GPU to implement our parallel model with the compute unified device architecture (CUDA) programming interface. Under CUDA, a GPU works as a co-processor of a conventional CPU, under the SIMT (*Single Instruction, Multiple-Thread*) architecture [26]. The SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. It is based on the concept of kernels which are functions written in C executed in parallel by a given number of CUDA threads. These threads will be launched onto GPU's streaming multi-processors and executed in parallel. More details about CUDA can be found in [26], [27].

5.1 GPU-based Algorithm

From the hardware point of view, GPU multiprocessor is based on thread-level parallelism to maximize the exploitation of its functional units. When adapting the cellular model to GPU implementation, we employ CUDA threads, as the processing units, to handle cells in parallel, and use CPU (host code) for flow control and the entire thread synchronization. The main CUDA algorithm is shown in Algorithm 1.

Algorithm 1 : CUDA algorithm

```
1: Initialize input disparity map, neural network, and cellular matrix;
2: Calculate cells' density values;
3: Find the max cell density value;
4: Calculate cells' activated probabilities;
5: for ite ← 0 to max_ite do
6:    if ite % memory_reuse_set_rate == 0 then
7:       Random number generation;
8:    end if
9:    if ite == 0 || ite % cell_refresh_rate == 0 then
10:      Refresh cells;
11:   end if
12:   Parallel SOM process;
13:   Modify SOM parameters;
14: end for
15: Save results;
```

Underscored lines 2, 4, 7, 10, and 12 are implemented with CUDA kernel functions that will be executed by GPU threads in parallel. In line 4, the cells' activation probabilities are computed according to the activation formula of Equation 6. In each iteration of the program, each cell needs two random numbers: one is used for cell activation and the other is used for roulette wheel extraction of input point in activated cell. With respect to the large scale input instances with huge cellular matrix and numerous iterations, the random numbers generated via kernel function in line 7 are stored in a fixed size area due to the limited GPU global memory. Every time these random numbers are used out, a new set of random numbers are generated at the beginning of the next iteration, depending on a constant rate factor called *memory_reuse_set_rate*. The random number generators we use in line 7 are from Nvidia CURAND library [26]. Each cell has data structures where are deposited data of the number and indexes, in the neural network, of the neurons this cell contains. These data may change during each iteration, but it appears that it can be sufficient to make the refreshing based on a refresh rate coefficient called *cell_refresh_rate*. Hence, the neurons' locations are moved in the plane at each single iteration, whereas the indexes in cells are refreshed based on a lower rate.

The parallel SOM process of line 12 in Algorithm 1 is further illustrated by Algorithm 2. Each cell needs to locate its position in the cellular matrix by its *threadId* and *blockId* [26]. During the parallel spiral search of each thread, we introduce a *spiral_search_range* (SSR) parameter to control the search scope. As shown in the bottom part of Fig. 6, the maximum number of cells a thread would search equals $(SSR \times 2 + 1)^2$. The original spiral search corresponds to setting SSR

Algorithm 2 : GPU parallel SOM process

1: Locate cell position associated to current thread
2: Check if the cell is activated;
3: **if** the cell is activated **then**
4: Perform local roulette wheel pixel extraction in the cell;
5: Perform spiral search within a certain range to find the winner neuron;
6: Modify positions of the winner neuron and its neighbors;
7: **end if**

infinite. All the neurons' locations are stored in GPU global memory which is accessible to all the threads. Like all the multi-threaded applications, different threads may try to modify a same neuron's location at the same time, which causes race conditions. This kind of situation only occurs in areas near cell boundaries where, for example, Thread A (responsible for Cell A) needs to move its winner neuron's neighborhood neuron—Neuron X, which is located in adjacent Cell B, while Thread B (responsible for Cell B) just needs to move Neuron X at the same time. In order to guarantee a coherent memory update in this situation, we use the CUDA atomic function which performs a read-modify-write atomic operation without interference from any other threads [26].

5.2 Parallelism Control

In order to achieve high performance, parallelism tuning is essential. It can be made in two ways from perspectives of both the cellular model and GPU running configuration.

5.2.1 Parallelism Parameters of Cellular Model

As discussed in Subsection 4.1, the size of the two-dimensional cellular matrix is $W/co \times H/co$, where $W \times H$ is the size of input disparity map and co is a constant factor used to control the degree of parallelism. This is the first parallelism parameter of the cellular model. The second parameter is the coefficient δ in Equation 6, which is used to adjust the degree of activity of cells. As a result, we can use the first parameter to control the number of cells in the cellular matrix and use the second parameter to adjust each cell's activation probability.

5.2.2 Kernel Decomposition for Parallel SOM

For the sake of GPU computing resource exploitation, it is necessary to keep the multiprocessors as active as possible. Latency hiding depends on the number of active warps per multiprocessor, which is determined by the number of threads per block and the total number of threads. In general, threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps [26]. However, this principle should be compromised, giving due consideration to the warp divergence [26], [27] exists in our kernel functions. In order to analyze the warp divergence issue and minimize its performance penalty, we designed two versions of parallel SOM process: the single-kernel version and the multiple-kernel version. A depth look at Algorithm 2 indicates that the three steps of line 4, 5, and 6 are actually independent, which brings possibility for kernel decomposition. Besides putting these three steps together into a whole kernel as the single-kernel version, we also implement each of them by an individual kernel in the multiple-kernel version. This kind of kernel decomposition allows us to set different kernel launch configurations for different steps of the parallel SOM process with separate kernels.

5.3 Memory Management

5.3.1 Memory Coalescing

For CUDA applications, accessing data in the global memory is critical to the performance. One main reason is the memory coalescing issue. In the GPU hardware, at any clock cycle, each multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data. Global memory bandwidth is most efficient when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32, 64, or 128 bytes. In this favorable case, the hardware coalesces all memory accesses into a consolidated access to consecutive DRAM locations. Otherwise, a separate memory transaction is issued for each thread and throughput is reduced.

In our implementation of the cellular model, four arrays are employed to store, for each cell, density value, activation probability, number of neurons, and 2-dimensional neuron identifiers in the network, respectively. For the first three arrays, the size is the number of cells in the cellular matrix and the element for each cell is close to each other. Note that all threads in the same warp have adjacent thread indices. For the adjacent threads assigned to the adjacent cells, memory accesses to the first three arrays are naturally coalesced. In the case of the fourth array, we designed two versions as shown in Fig. 7. In Version 1, the 2-dimensional coordinate components (X_G and Y_G) of neurons for each cell are stored respectively in *CellNeuronId_x* and *CellNeuronId_y*. The elements of each cell are interlaced in such a way that neighbouring cells access neighbouring positions in the same searching step, with the whole step-by-step searching process beginning from the first neuron in the cell and ending to the last neuron. Consequently, in every searching step, the memory accesses of adjacent threads assigned to the adjacent cells are coalesced. In

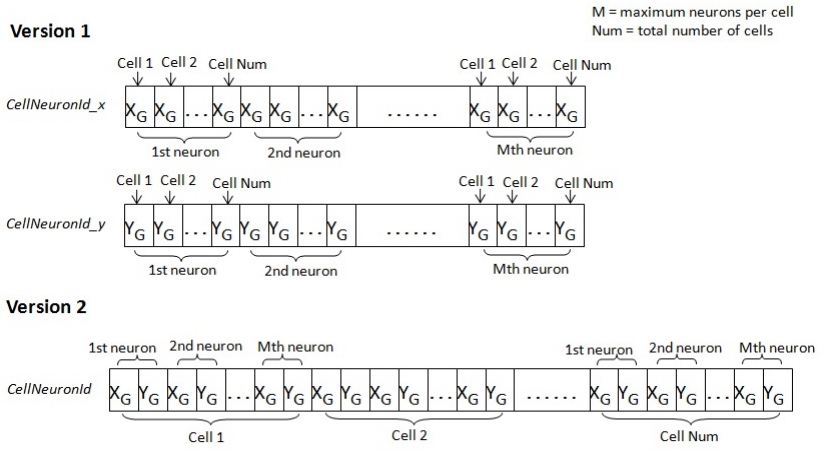


Fig. 7: Two memory organization versions of the array for 2-dimensional neuron identifiers of cells.

Version 2, each neuron's pair of 2-dimensional coordinate components are put together and coordinates of all the neurons in the same cell are arranged close to each other. As a result, in the array of *CellNeuronId*, all the data needed for every cell during the whole step-by-step searching process are located in contiguous address space. In both versions the data structure assumes a fixed maximum number (M in Fig. 7) of neurons per cell. Also, threads access cells in a competitive manner and GPU version requires atomic operations when different neurons try to update a same cell at the same time.

5.3.2 Shared Memory and Texture Memory

Two other kinds of GPU memories, which are commonly used for additional speedups, are shared memory and texture memory. The shared memory, accessible only to threads in the same block, resides physically on the GPU as opposed to residing in off-chip DRAM. Because of this, the latency to access shared memory tends to be far lower than to access global memory directly. In our case, however, the shared memory seems unsuitable for acceleration for the following two reasons: (1) The shared memory is preferred to buffer the data which need to be accessed many times at the same position, like in the case of summation reduction. While in our cellular model, for each thread, the data stored at a given position is only read or written once in most of the time, like the extracted pixel's coordinate information; (2) The shared memory is ideal to store intermediate computation results that are shared and reused by neighboring threads from the same warp. While in our cellular model, the operation of each thread, among many threads launched at every parallel iteration, is essentially independent from each other.

The texture memory also has on-chip caches designed for graphics applications where memory access patterns exhibit a great deal of spatial locality [26]. In our application, during the parallel learning process of each cell, memory accesses into the array with neurons' positions are promising to be accelerated by texture memory in that the winner neuron, as well as its nearby neurons in a certain neighborhood in the 2-dimensional network, need to be moved. Therefore, we bind this part of global memory accesses, which is with spatial locality, to textures, trying to benefit from the texture cache.

6 Experimental Analysis

During our experimental study, we have used the following platforms:

- *On the CPU side:* An Intel(R) Core(TM) 2 Duo CPU E8400 processor running at 2.67 GHz and endowed with four cores and 4 Gbytes memory. It is worth noting that only one single core executes the SOM process in our implementation.
- *On the GPU side:* A Nvidia GeForce GTX 570 Fermi graphics card endowed with 480 CUDA cores (15 streaming multi-processors with 32 CUDA cores each) and 1280 Mbytes memory.

TABLE 1: Experiment parameters.

α_{init}	α_{final}	σ_{init}	σ_{final}	MNPC[a]	MRSR[b]	δ
1	0.01	$24/12^c$	1	2000	1000	1

[a] maximum number of neurons per cell. [b] *memory_reuse_set_rate*.
[c] 12 is only for the experiments of Subsection 6.4.

Some parameters are fixed among the whole experimentation, as shown in Table 1. Parameter values for SOM were adjusted after a preliminary round of experiments which are not reported here. All the inputs of disparity maps we have used in the section are from the Middlebury Stereo Datasets [17], [28]–[31]. All the experimental results reported in the following subsections are mean values of 10 runs.

6.1 Experiment for Memory Management

In this subsection, we verify the effectiveness of our memory management strategy by carrying out experimental comparisons between the two memory

organization versions (MOV) of Fig. 7. The test is done with inputs of four small size disparity maps (*tsukuba, venus, teddy, cones*), one medium size (*conesM*), and one large size (*conesL*). In order to eliminate influences of other irrelevant factors, we set same experimental parameters of all the six inputs. We only consider the most time-consuming part of the program, the parallel SOM process which runs *max_ite* times, as shown in Algorithm 1, because compared with it, time taken by other parts is quite trivial according to our trial tests. During the experiment, we configure the SOM kernel with 32 threads per block, which is the minimal number to fully fill a warp. The running time of SOM kernel with six different inputs for 1500 iterations is reported in Fig. 8.

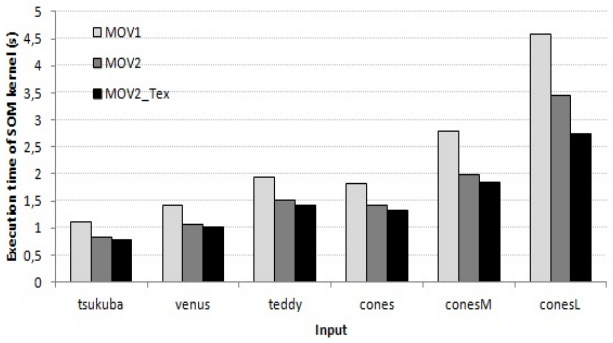


Fig. 8: Performance comparison of different memory organization versions.

Results show that SOM kernel with memory organization version 2 (MOV2) runs faster than with memory organization version 1 (MOV1), for each of the six tested inputs. We think this is because MOV2 can benefit from L1/L2 cache of GPU cards with compute capability 2.x. In the array of *CellNeuronId* of MOV2, all the data needed for a cell/thread during the whole step-by-step searching process are located in contiguous address space. These data are cached through the access of the first searching step, and for the following steps, data are read directly from cache. In MOV1, although simultaneous memory accesses of a same searching step for adjacent cells/threads are coalesced, data of adjacent steps for one single cell/thread are not cached because it is separated step-by-step with gap distance of *Num* (total number of cells). This point is highlighted by CUDA Profiler's execution profiling whose results show that L1 cache hit ratio is about 80% in MOV2 while it is only about 20% in MOV1. According to our experiment results, we think that, compared with coalesced memory accesses of adjacent cells/threads in each simultaneous searching step, the performance gain from caching data among different steps of one single cell/thread is more fruitful.

In our application with MOV2, after binding the array of neurons' positions to texture memory during the learning process with spatial locality, we can gain further performance improvement especially for the large size input, as shown by MOV2_Tex in Fig. 8.

6.2 Experiment for Parallelism Control

In this subsection, we consider different kinds of factors and parameters associated with the cellular model's parallelism control, and evaluate different influences, on both running time and result quality, with configuring combinations of various parameters. Three main tests are carried out with parallelism degree control parameter co set to 10, 20, and 30 successively. During each test, we run both the two versions of parallel SOM process – the single-kernel version and the multiple-kernel version – with seven block sizes (number of threads in every CUDA block) for each kernel function's launch configuration: 1, 8, 16, 32, 64, 128, 256. Test results with input of *conesM* disparity map (900×750 pixels) for 1500 iterations are reported in Fig. 9.

In Fig. 9, running time of single-kernel SOM (SK) and multiple-kernel SOM (MK) as well as its three consisting kernel functions: kernel for extraction step (MK_E), kernel for spiral search step (MK_S), and kernel for learning step (MK_L) are reported. Note that the values of MK are simply the sum of values of MK_E, MK_S, and MK_L. With co set to 10, the fastest block size configuration for SK, MK_E, MK_S, MK_L are 64, 256, 64, 64 respectively; with co set to 20, the fastest block size configuration for SK, MK_E, MK_S, MK_L are 8, 16, 8, 16 respectively; with co set to 30, the fastest block size configuration for SK, MK_E, MK_S, MK_L are 1, 8, 1, 8 respectively.

In the GPU computing point of view, our cellular model with more cells (smaller co value) tends to better explore parallel computing resources since its fastest block size is a multiple of warp size. For the two versions with co value of 20 and 30, however, they run slower when block size is set to multiples of warp size than smaller ones. We think the main reason here is warp divergence. In each of the three steps of SOM process, there exist possibilities of divergent branches taken by different threads: at the first step, they are in the random roulette wheel point extraction procedure; at the second step, they are in the spiral search procedure whose stop point is unpredictable; at the third step, they are in the learning procedure when dealing with neuron neighborhoods cut by network boundaries. Hence, the more threads are assigned together into a warp unit to execute in lockstep on a streaming multiprocessor (SM), the higher frequency that warp divergence will occur. Note that all the treads in a warp are from one same block and have consecutive *threadIdx* values. The warp divergence issue gets severe particularly for the kernels with spiral search procedure (SK and MK_S) when co is big. This is because smaller number of cells correspondingly causes each cell contains more neurons in average,

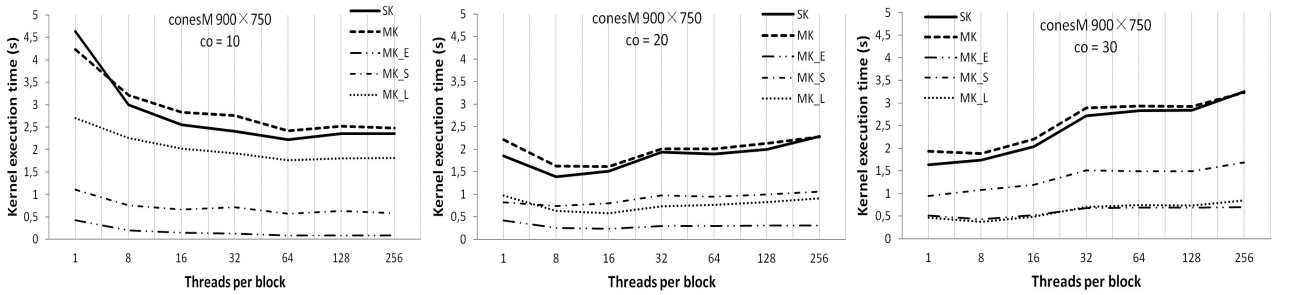


Fig. 9: Comparative results of single-kernel SOM (SK) and multiple-kernel SOM (MK) with different parallelism control parameters.

which enhances frequency of occurrence of the scenario that different threads in a warp stop searching after checking different numbers of neurons.

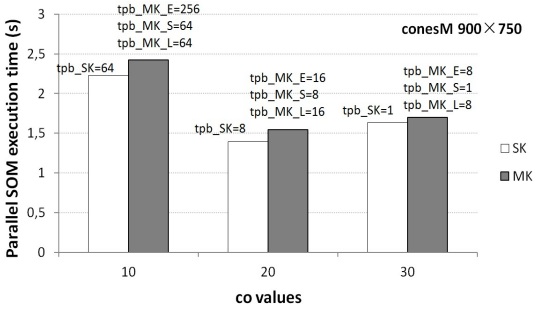


Fig. 10: Performance comparison between single-kernel version and multiple-kernel version of the parallel SOM process.

According to the fastest block size configurations found in the previous experiment, we compare performance between single-kernel version and multiple-kernel version of the parallel SOM process. In every test under a different *co* value, we set the block sizes of MK_E, MK_S, MK_L to the fastest ones and compare their sum running time with SK whose block size is also set to the fast one. Results are reported in Fig. 10. The multiple-kernel version runs slower in all the three cases. This is caused by additional global memory accesses introduced in the multiple-kernel version. Specifically, we have to store the extracted points from MK_E into global memory in order to let MK_S use them. Similarly, we have to store the closest neuron found by MK_S into global memory in order to let MK_L use them. As to the comparison among three different *co* values, the parallel SOM process runs fastest when *co* is set to 20, with either single-kernel version or multiple-kernel version. It runs slower when *co* is set to 30 due to the poor exploitation of GPU parallel computing resources, while the fact that it also runs slower when *co* is set to 10 can be explained by the employment of too many cells, hence too many threads need to execute. As a result, we can fairly say that in this experiment, the cellular model is under-parallelized when *co* is set to 30 whereas it is over-parallelized when *co* is set to 10.

6.3 Running Time/Quality Tradeoff

In our GPU implementation, spiral search range (SSR) and cell refresh rate (CRR) are two important parameters which can impose influences on both running time and result quality. If we set a bigger SSR meaning that the spiral search will proceed in a larger amount of cells, or if we set a smaller CRR meaning that the cellular matrix will be refreshed after fewer times of iterations, the running time will increase while the result quality will improve. Trying to find a good compromise between computation time and result quality, we have carried out experiments with different value combinations of these two parameters. Three tests are performed with a small size input, a medium size one, and a large size one respectively. During all the tests, the single-kernel version of parallel SOM is employed. Parallelism degree control parameter *co* is set to 20. SSR is set to 2, 3, 4, 5, 10 and infinite, successively, and for each SSR value, CRR is set to 1, 10, 20, 30, 40, 50 and 100, successively. Time/quality results of the 42 combinations for each test are demonstrated in Fig. 11.

Compared with SSR values, different CRR values have fairly slighter effects on running time in all three tests, which indicates that we are allowed to set a small CRR value in order to obtain better results. For the setting of SSR value, a wise choice should be confining it under a certain upper limit value because after SSR exceeds this value, result quality stops turning better. For example, according to the test of small size input, setting SSR to 4, 5 and 10 allows getting similar best result quality as the original spiral search with SSR being infinite. Therefore, the upper limit value of SSR for this small size input should be 4. Similarly, this value for the tested medium size input and tested large size input should be 3 and 5 respectively.

Another important parameter concerned with running time and quality tradeoff is the number of iterations. We perform another experiment trying to provide some insight on the determination of this parameter. Two tests are carried out with *cones* and *teddy* disparity maps. In each test, we successively set 15 different numbers of iterations form 100 to 1500

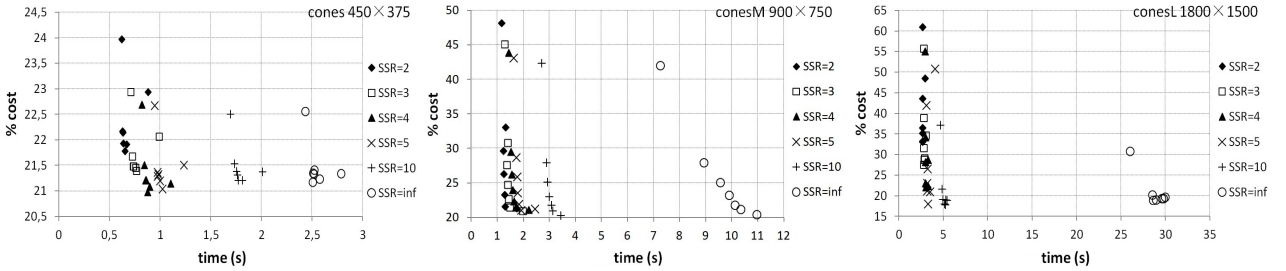


Fig. 11: Experiment of time/quality compromise regarding SSR and CRR. From left to right: small size, medium size, large size.

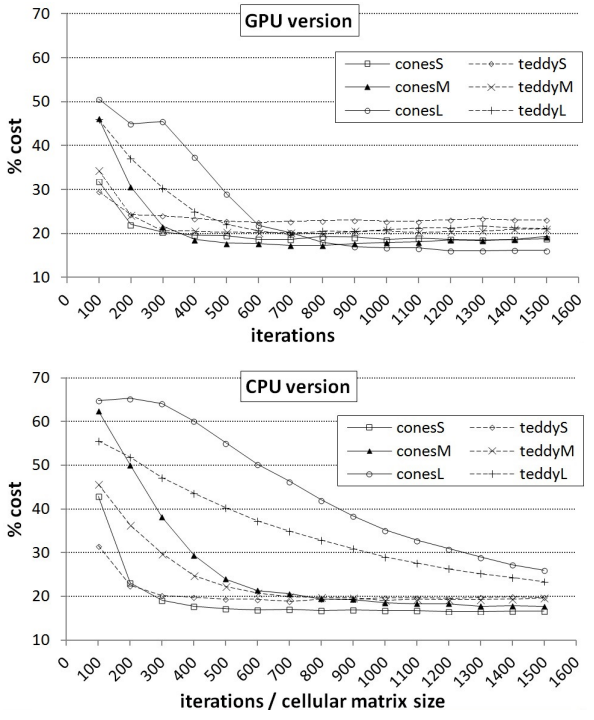


Fig. 12: Relationship between $\%cost$ value and iteration number.

for GPU version and correspondingly 15 different numbers of iterations form 100 $\times$ *cellular matrix size* to 1500 $\times$ *cellular matrix size* for the counterpart CPU version. The relationship between $\%cost$ value and iteration number is shown in Fig. 12. The setting of 1500 iterations for GPU version (accordingly 1500 $\times$ *cellular matrix size* for CPU version) is convergent but not necessarily optimal, for all the test cases except for the two large size inputs with CPU version. The optimal value of the number of iterations, which should be the smallest convergent value, varies among different inputs and different parameter settings. Consequently, it is impracticable to set a unified optimum for all inputs. The optimal parameter setting of our method is an open question, like other optimization algorithms.

6.4 Results of Disparity Maps at Different Scales

In this subsection, we firstly perform experiments using six small size disparity maps, with a smaller spiral search range (SSR=3). Results are shown in Table 2. Note that both the number of sequential iterations and the CRR value for CPU version of each test are set to its corresponding GPU version values multiplied by *cellular matrix size*. The acceleration factor, which is the ratio of CPU version's running time by GPU, varies from 5.84 (*tsukuba*) to 8.33 (*aloe*). Experimental results in the first set of Table 2 show that our proposed parallel model can provide near real-time performance, by its GPU implementation, for structured mesh generation of small size disparity maps. The result quality is very similar to its counterpart sequential CPU version in average.

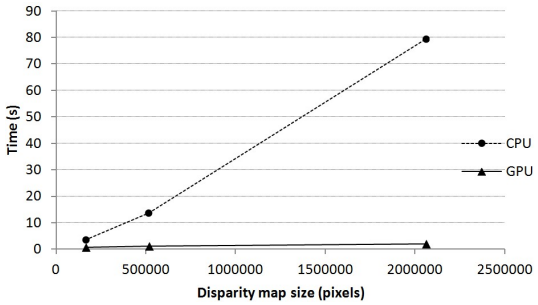


Fig. 13: Comparative results between CPU and GPU for the time/size relationship.

We also study the relationship between running time and input size of our GPU implementation and its counterpart CPU version, by carrying out experiments with four disparity maps at small, medium and large scales, respectively. Inputs, parameter settings and results are all shown in the last three sets of Table 2. The average acceleration factors of the three sets are 5.49, 12.68 and 39.74 respectively, as the input size grows. Based on the average data of each tested set, the relationship between running time and input size is depicted in Fig. 13. We can note that the running time of GPU version increases in a linear way with a very weak increasing coefficient which is almost zero, when compared to the CPU version. Meanwhile, the result qualities of the two versions are quite similar in average, for each tested set with different sizes. Results in Fig. 13 verify our proposed cellular model's characteristic of "massive parallelism" in a practical way by our GPU implementation. We consider that

TABLE 2: Experimental data of four sets of disparity maps at different scales.

input image	image size	grid size (g)	cellular matrix size (co)	SSR	CRR^g	iterations^g	time (s) GPU	time (s) CPU	%cost GPU	%cost CPU	AF[1]
tsukuba	384 × 288	64 × 48 (6)	20 × 15 (20)	3	20	1500	0.32	1.87	25.89	25.02	5.84
rocks1	425 × 370	71 × 62 (6)	22 × 19 (20)	3	20	1500	0.34	2.28	17.94	18.73	6.71
aloe	427 × 370	71 × 62 (6)	22 × 19 (20)	3	20	1500	0.30	2.50	27.15	28.70	8.33
venus	434 × 383	72 × 64 (6)	22 × 20 (20)	3	20	1500	0.32	2.40	21.55	19.62	7.50
teddy	450 × 375	75 × 63 (6)	23 × 19 (20)	3	20	1500	0.36	2.41	18.45	20.04	6.69
cones	450 × 375	75 × 63 (6)	23 × 19 (20)	3	20	1500	0.35	2.45	17.24	16.69	7.00
Average	**154926**						**0.33**	**2.32**	**21.37**	**21.47**	**7.03**
rocks1S	425 × 370	71 × 62 (6)	22 × 19 (20)	4	20	1500	0.66	3.32	20.03	18.91	5.03
aloeS	427 × 370	71 × 62 (6)	22 × 19 (20)	4	20	1500	0.51	3.56	27.45	28.02	6.98
conesS	450 × 375	75 × 63 (6)	23 × 19 (20)	4	20	1500	0.67	3.52	18.95	16.59	5.25
teddyS	450 × 375	75 × 63 (6)	23 × 19 (20)	4	20	1500	0.69	3.45	18.19	19.91	5.00
Average	**163185**						**0.63**	**3.46**	**21.16**	**20.86**	**5.49**
rocks1M	638 × 555	106 × 93 (6)	32 × 28 (20)	4	20	1500	1.05	8.33	21.65	18.37	7.93
aloeM	641 × 555	107 × 93 (6)	33 × 28 (20)	4	20	1500	0.57	9.36	26.39	26.90	16.42
conesM	900 × 750	150 × 125 (6)	45 × 38 (20)	4	20	1500	1.36	18.52	18.75	17.82	13.62
teddyM	900 × 750	150 × 125 (6)	45 × 38 (20)	4	20	1500	1.35	18.53	21.07	19.26	13.73
Average	**514961**						**1.08**	**13.69**	**21.97**	**20.59**	**12.68**
rocks1L	1276 × 1110	213 × 185 (6)	64 × 56 (20)	4	20	1500	2.18	47.36	21.73	19.53	21.72
aloeL	1282 × 1110	214 × 185 (6)	65 × 56 (20)	4	20	1500	0.95	53.19	25.64	24.72	55.99
conesL	1800 × 1500	300 × 250 (6)	90 × 75 (20)	4	20	1500	2.74	107.10	17.27	25.93	39.09
teddyL	1800 × 1500	300 × 250 (6)	90 × 75 (20)	4	20	1500	2.13	110.25	25.98	23.63	51.76
Average	**2059845**						**2.00**	**79.48**	**22.66**	**23.45**	**39.74**

[1] acceleration factor. ^{g} GPU version.

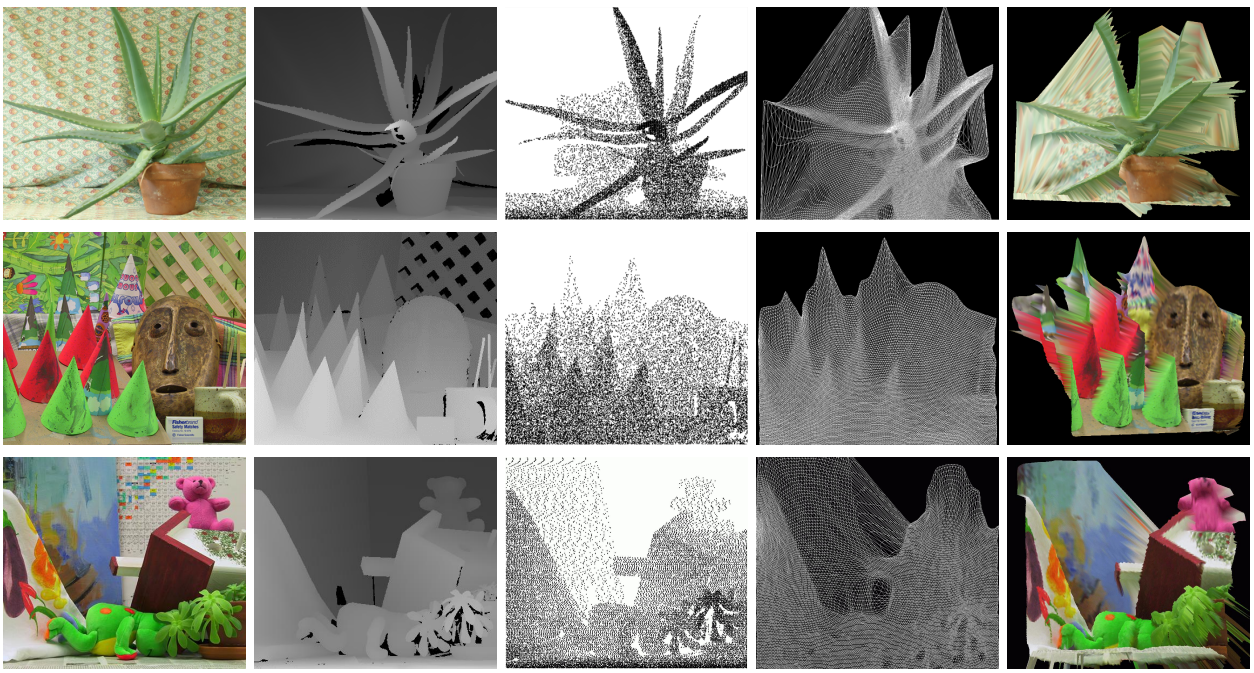


Fig. 14: Application snapshots. From top to bottom: *aloe*, *cones*, *teddy*. From left to right: color image, disparity map, density sampling, meshing, visualization in 3D space.

such results are encouraging when solving very large scale Euclidean NP-hard optimization problems.

6.5 Visualization of 3D Reconstruction

Fig. 14 displays snapshots of our stereo vision application dealing with *aloe*, *cones*, and *teddy*, respectively. In the first column are the color stereo images from left view and in the second column are their corresponding disparity maps. In the third column are samplings of disparity maps obtained by extracting 10000 points with the roulette wheel mechanism. This sample has been obtained after having removing the background small density values from the disparity map, and after augmenting the contrast in it. Then, objects that are close to the camera have higher density values. In the fourth column are the adaptive 2D meshes obtained by SOM algorithm with the modified disparity maps. In the second column, brighter regions are nearer to the camera view point, and in the third column such nearer regions present higher density of extracted points, whilst in the fourth column the adapted grid presents higher density of neural network nodes on

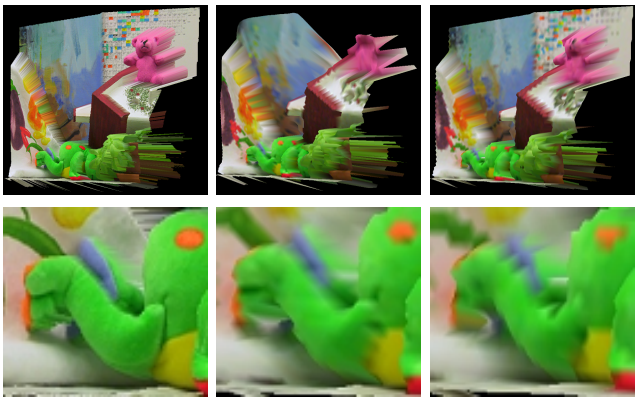


Fig. 15: An example of higher resolution for closer objects. From left to right: uniform mesh with high resolution (450×375), adapted mesh (75×63), uniform mesh (75×63). First row: global scene. Second row: zoom on a part of a closest object of the scene.

such regions, with respect of the topology of the scene. The fifth column shows the surface reconstruction in 3D space obtained by using the adapted mesh which can be seen as a compressed representation of the 3D surface, such that objects close to the camera have higher resolution and their details are more finely represented.

Fig. 15 shows an example of higher resolution for closer objects, where different visualizations of 3D reconstructions are respectively from high resolution uniform mesh of original image, adapted mesh of low resolution, and uniform mesh of low resolution, with a compression rate of 36. The zoom comparison illustrates the resolution around a closest object. The adapted mesh presents a higher resolution for the closest object in the scene than the uniform reduced mesh.

7 Conclusion

We have proposed a massively parallel cellular model for structured mesh generation of stereo disparity maps. The structured mesh generation is presented as an NP-hard balanced clustering problem in the plane, and the Kohonen's self-organizing map is used as a heuristic to deal with it. Our approach is based on control distribution and data decomposition. With independent cells, the distributed computation does not depend on any central control. The model is dimensioned with $O(N)$ for both processing units and memory size, where N is the input size. We have contributed with a GPU implementation with CUDA, to verify the effectiveness and efficiency of the proposed model. The parallelism of SOM process can be achieved either by using a single kernel function, to avoid additional global data communications between threads, or by employing separate kernels for different steps, to improve degree of parallelism controllability. Experimental results show that our GPU implementation is able to provide near real-time performance for the structured mesh generation of small size disparity maps, and the average acceleration factor to its counterpart sequential CPU version varies from 5.49 for small size inputs to 39.74 for large size inputs. The running time of our GPU implementation increases in a linear way with a very weak increasing coefficient which is nearly zero, when compared to the CPU version.

Future work should deal with verification of effectiveness of the model as the number of physical cores augments. More precisely, we should verify the possibility of designing an ideally near constant time algorithm, for bounded or uniform distributions, when the number of handled physical cores really increases as the instance size increases. It should be of interest also to study more CUDA programming techniques, for a better memory management, or the use of multiple GPUs, in order to achieve faster real-time computing on massively parallel platforms and integrate applications to real-time environments like intelligent vehicle systems.

References

[1] T. Kohonen, "Clustering, Taxonomy, and Topological Maps of Patterns," *Proceedings of the 6th International Conference on Pattern Recognition*, October 1982.

[2] ——, "Self-Organization Maps and associative memory," *Spring Verlag*, 2001.

[3] P. George, "Génération automatique de maillages. ," *Applications aux méthodes d'éléments finis,Masson, RMA 16*, 1991.

[4] M. Bern and D. Eppstein, "Mesh generation and optimal triangulation," *Computing in Euclidean geometry*, vol. 1, pp. 23–90, 1992.

[5] G. F. Carey, *Computational grids: generation, adaptation, and solution strategies*. CRC Press, 1997.

[6] S. J. Owen, "A survey of unstructured mesh generation technology." in *IMR*, 1998, pp. 239–267.

[7] J. P. Pons, F. Segonne, J. D. Boissonat, L. Rineau, M. Yvinec, and R. Keriven, "High-Quality Consistent Meshing of Multi-Label Datasets," *;Information Processing in Medical Imaging*, pp. 198–210, 2007.

[8] F. Tombari, S. Mattoccia, and L. D. Stefano, "Segmentation-based adaptive support for accurate stereo correspondence," *Proc PSIVT*, pp. 427–438, 2007.

[9] F. de Goes, S. Goldenstein, and L. Velho, "A simple and flexible framework to adapt dynamic meshes," *Computer & Graphics*, pp. 141–148, 2008.

[10] E. Bonomi, "Generation of structured adaptative grids based upon molecular dynamics," *Comptes rendus des journées d'électronique, Presses Polytechniques Romandes, Lausanne*, 1989.

[11] O. Sarzeaud, Y. Stéphan, and C. Touzet, "Finite Element Meshing using Kohonen's Self-Organizing Maps," *International Conference on Artificial Neural Network*, juin 1991.

[12] J. C. Créput, T. Lissajoux, and A. Koukam, "A connexionnist approach to the hexagonal mesh generation," 2000.

[13] R. Schneiders, "Algorithms for Quadrilateral and Hexahedral Mesh Generation," *Proceedings of the VKI Lecture series on Computational Fluid Dynamic*, pp. 2000–2004, 2000.

[14] Y. Zhang and C. Bajaj, "Adaptive and quality quadrilateral/hexahedral meshing from volumetric data," *Comput. Methods Appl. Mech. Eng.,*, pp. 942–960, 2006.

[15] N. Megiddo and K. J. Supowit, "On the complexity of some common geometric location problems," *SIAM journal on computing*, vol. 13, no. 1, pp. 182–196, 1984.

[16] U. Pferschy, R. Rudolf, and G. J. Woeginger, "Some geometric clustering problems," *Nord. J. Comput.*, vol. 1, no. 2, pp. 246–263, 1994.

[17] D. Scharstein and R. Szeliski, "Middlebury Stereo Datasets." 2012, http://vision.middlebury.edu/stereo/.

[18] H. Lu, Y. Wu, and S. Chen, "A new method based on SOM network to generate coarse meshes for overlapping unstructured multigrid algorithm," *Applied Mathematics and Computation*, pp. 353–360, 2003.
[19] R. L. M. E. do Rego, A. F. R. Araujo, and F. B. d. L. Neto, "Growing Self-Organizing Maps for Surface Reconstruc tion from Unstructured Point Clouds ," *Int. Joint Conf. Neural Netw.*, pp. 1900–1905, 2007.
[20] O. Nechaeva, "Using Self Organizing Maps for 3D surface and volume adaptive mesh generation," *http://www.intechopen.com/books/self-organizing-maps/using-self-organizing-maps-for-3d-surface-and-volume-adaptive-mesh-generation*, 2010.
[21] A. C. Castilla and N. G. Blas, "Self–organizing map and Cellular automata combined technique for advanced mesh generation in urban and architectural design," *International Journal "Information Technologies and knowledge"*, vol. 2, 2008.
[22] M. Yoon, I. P. Ivrissimtzis, and S. Lee, "Self-organising maps for implicit surface reconstruction," in *TPCG*. Citeseer, 2008, pp. 83–90.
[23] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, and R. Hurley, "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA," vol. 341, no. 1. IOP Publishing, 2012, pp. 012–018.
[24] M. Yoshimi, T. Kuhara, K. Nishimoto, M. Miki, and T. Hiroyasu, "Visualization of Pareto Solutions by Spherical Self-Organizing Map and It's Acceleration on a GPU," *Journal of Software Engineering and Applications*, vol. 5, no. 3, 2012.
[25] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal expected-time algorithms for closest-point problems," *Computer Science Department*, p. 2451, 1979.
[26] NVIDIA, "CUDA C Programming Guide 4.2, CURAND Library, Profiler User's Guide." 2012, http://docs.nvidia.com/cuda.
[27] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.
[28] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International journal of computer vision*, vol. 47, no. 1-3, pp. 7–42, 2002.
[29] ——, "High-accuracy stereo depth maps using structured light," in *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, vol. 1. IEEE, 2003, pp. I–195.
[30] D. Scharstein and C. Pal, "Learning conditional random fields for stereo," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.
[31] H. Hirschmuller and D. Scharstein, "Evaluation of cost functions for stereo matching," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.

Hongjian Wang received his M.S. degree in Computer Science from School of Computer Engineering and Science, Shanghai University, China, in 2012. He is currently a Ph.D. candidate at the Systems and Transportation Laboratory (SeT) of the Research Institute on Transportation, Energy and Society (IRTES), University of Technology of Belfort-Montbéliard (UTBM), France. His current research interests include metaheuristic optimization, neural networks, computer stereo vision and parallel computing.

Naiyu Zhang received his M.S. degree in System Optimization and Security from University of Technology of Troyes (UTT), France, in 2010. He received his Ph.D. in Computer Science from the University of Technology of Belfort-Montbéliard (UTBM), France, in 2014. His current research interests include metaheuristic optimization, neural networks, computer stereo vision and parallel computing.

Jean-Charles Créput received his Ph.D. in Computer Science from University of Paris 6, France, in 1997. He defended his Habilitation in November 2008 at the University of Bourgogne, France. He is currently an associate professor in Computer Sciences and Engineering at the University of Technology of Belfort-Montbéliard (UTBM), France, and performs research activity at the Systems and Transportation Laboratory (SeT). His current research interests include evolutionary algorithms, neural networks and multi-agent systems applied to telecommunications and intelligent transportation services.

Julien Moreau received his M.Sc. degree in audio and image system engineering "Master ISIS" from the audiovisual department of the University of Valenciennes and Hainaut-Cambrésis, Valenciennes, France, in 2011. He is currently a Ph.D. candidate at the Systems and Transportation Laboratory (SeT) of the Research Institute on Transportation, Energy and Society (IRTES), University of Technology of Belfort-Montbéliard (UTBM), France. His research interests cover image processing for stereovision, fisheye vision, cameras calibration and 3D modeling of urban structures.

Yassine Ruichek received respectively his Ph.D. in control and computer engineering and Habilitation à Diriger des Recherches (HDR) in physic science from the University of Lille, France, in 1997 and 2005.

He was an assistant researcher and teacher from 1998 to 2001. From 2001 to 2007 he was an associate professor at the University of Technology of Belfort-Montbéliard (UTBM), France. Since 2007, he is a full professor at UTBM. He conducts research activities in multi-sources/sensors fusion for environment perception and localization. His research interests are computer vision, data fusion and pattern recognition. Since 2012, he is the head of the Systems and Transportation Laboratory (SeT) of the Research Institute on Transportation, Energy and Society (IRTES).