Cellular GPU Models to Euclidean Optimization Problems : Applications from Stereo Matching to Structured Adaptive Meshing and Traveling Salesman Problem

Naiyu Zhang

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

Cellular GPU Models to Euclidean Optimization Problems

Applications from Stereo Matching to Structured Adaptive Meshing and Travelling Salesman Problem

Naiyu ZHANG

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE TECHNOLOGIE BELFORT-MONTBÉLIARD

N° | X | X | X |

THÈSE présentée par

Naiyu ZHANG

pour obtenir le

Grade de Docteur de
l'Université de Technologie de Belfort-Montbéliard

Spécialité : **Informatique**

Cellular GPU Models to Euclidean Optimization Problems

Applications from Stereo Matching to Structured Adaptive Meshing and Travelling Salesman Problem

Soutenue le 2 Décembre 2013 devant le Jury :

Abdelaziz BENSRHAIR	Rapporteur	Professeur des Universités à l'Institut National des Sciences Appliquées de Rouen
René SCHOTT	Rapporteur	Professeur des Universités à l'Université de Lorraine, Nancy
Lhassane IDOUMGHAR	Examinateur	Maître de Conférences HDR à l'Université de Haute Alsace
Behzad SHARIAT	Examinateur	Professeur des Universités à l'Université Claude Bernard de Lyon I
Jean-Charles CREPUT	Directeur	Maître de Conférences HDR à l'Université de Technologie de Belfort-Montbéliard
Abderrafiâa KOUKAM	Examinateur	Professeur des Universités à l'Université de Technologie de Belfort-Montbéliard
Yassine RUICHEK	Examinateur	Professeur des Universités à l'Université de Technologie de Belfort-Montbéliard

ACKNOWLEDGMENTS

At the final stage of preparing the documentation of my work, this section gives me the opportunity to express my gratitude to all the people who supported and assisted me.

First of all, my research would not have been possible without the generous funding of CSC-China Scholarship Council programme[1].

[1] For more information, please refer to http://en.csc.edu.cn

CONTENTS

1

General Introduction

1.1/ Context

It is now a general tendency that computers integrate more and more transistors and processing units into a single chip. Personal computers are currently multi-core platforms. This happens in accordance with the Moore's law that states that the number of transistors on integrated circuits doubles approximately every two years. At the same time, personal computers most often integrate graphic acceleration multiprocessor cards that become more and more cheaper. This is particularly true for Graphics Processing Units (GPU) which were originally hardware blocks optimized for a small set of graphics operations. Hence, the concept of GPGPU, that stands for general-purpose computing on graphics processing units, emerges by recognizing the trend of employing GPU technology for not only graphic applications but also general applications. In general, the graphic chips, due to their intrinsic nature of multi-core processors and being based on hundreds of floating-point specialized processing units, make many algorithms able to obtain higher performances than usual Central Processing Unit (CPU).

The main objective of this work is to propose parallel computation models and parallel algorithms that should benefit from the GPU's enormous computational power. The focus is put on the field of combinatorial optimization and applications in embedded systems and terrestrial transportation systems. More precisely, we develop tools in relation to Euclidean optimization problems in both domains of stereovision image processing and routing problems in the plane. These problems are NP-hard optimization problems. This work presents and addresses stereo-matching problem, balanced structured meshing problem of a data distribution, and also the well known Euclidean traveling salesman problem. Specific GPU parallel computation models are presented and discussed.

1.2/ Objectives and concern of this work

The major concern of the work could be summarized as:

Propose a computation model for GPU that allows
(i) massive GPU parallel computation for Euclidean optimization problems, such as image processing and TSP,
(ii) application in real-time context and/or to large scale problems within acceptable computation time.

In the field of optimization, GPU implementations are more and more studied to accelerate metaheuristics methods to deal with NP-hard optimization problems, and large size problems that can not be addressed efficiently by exact methods. Here, we restrict our attention to heuristics and metaheuristics. Often, they exploit natural parallelism of metaphors, such as ant colony algorithms, or genetic algorithms that present an inherent level of parallelism by the use of a pool of solutions to which can be applied simultaneous independent operations. A most studied example, is the computation of solution evaluations in a parallel way. Meanwhile, such methods are based on parallel duplication of the solution data for parallelism. According to a fixed memory size GPU device, it follows that the input problem data size should decrease with the increase of the population size, and hence with the increase of the number processing units used in parallel. Such population based approach is memory consuming if one wants to deal with large size problems and large size populations together, that are contradictory requirements. Other methods are local search or neighborhood search approaches that operate on a single solution by improving it, successively with a neighborhood search operator. In that case, most of the GPU approaches try to compute the neighborhood candidates in parallel and select the best candidate. Due to the size of neighborhoods which are generally large, such approaches are facing to a difficult implementation problem of GPU resources management. The number of threads could be very high. For example, a single 2-opt improvement move should require $O(N^2)$ evaluations, with N the problem size. How to assign such evaluations computation to parallel threads is a difficult question. For the moment of writing, it looks that such approaches have only addressed relatively small size problems, considering a very standard problem such as the Euclidean traveling salesman problem. For each problem at hand, the designer has to carefully assign neighborhood operations to computation resources, as threads and registers.

To contribute at the development of GPU approaches able to deal with large size problems, we restrict our attention to Euclidean optimization problems and address a different way to tackle data input at a low level of granularity. We follow the local dense approaches in image processing that simply assign a little part of the data to each computing unit. Pixels of an image constitute a cellular decomposition of the data that yields to a natural level of computational parallelism. To address different Euclidean optimization problems, we extend the model of cellular decomposition to neural networks topological map algorithms that operate by the multiplication of simple operations in the plane. Such operations produce, or make emerge, the required solution. The general approach that we retain for massive parallelism computation is cellular decomposition of the plane between a grid of cells, each one assigned to a given small and constant part of the input data, and hence to a single processing unit.

The cellular decomposition concerns input data of the problem. The approach differs from cellular genetic algorithms where processors are organized into a grid and where each one manages an independent solution. Such approach exploits data duplication parallelism, whereas our approach exploits data decomposition parallelism. An important

point, is that we are using such decomposition in accordance to the problem size, with a linear relationship, in order to address large size problems. We will illustrate that point on the Euclidean traveling salesman problem. Another point, is that we focus on a distributed and decentralized algorithm, with quite no intervention of the CPU during computation. It is worth noting that GPU local search methods often operate in a sequential/parallel way, where CPU may have a central role to prepare the next parallel computation for neighborhood move. We only consider methods where no solution transfers occurs between GPU and CPU during the course of the parallel execution. The main points are a low granularity level of data decomposition, together with distributed computation with no central control. The parallel cellular model does not prevent from using it into larger population based methods, or in combination with standard local search operators. We are investigating a complementary way of addressing some Euclidean optimization problems that can be embedded in more general strategies.

In this work, we are implementing two types of algorithms. They are the winner-takes-all (WTA) local dense algorithm for stereo-matching and the self-organizing map (SOM) neural network algorithm for structured meshing. The continuity of the method is illustrated on applications in the field of artificial vision, 3D surface reconstruction, that use both methods, and to Euclidean traveling salesman problem that uses the SOM.

The goal of a stereo-matching algorithm is to produce a disparity map that represents the 3D surface reconstructed from an image pair acquired by a stereo camera. We first study a stereo-matching method based on color filter array (CFA) image pairs. If a naive and direct GPU implementation can easily accelerate its CPU counterpart, the real-time requirement could be achieved. Then, starting from this basic GPU application, we investigate acceleration mechanisms to allow near real-time computation. Memory management appears to be a central factor for computation acceleration, whereas use of specific support region for matching and refinement steps improve quality substantially. Then, the parallel self-organizing map for compressed structured mesh generation is presented. Starting from a disparity map as input, the algorithm generates an hexagonal mesh that can be used as a compressed representation of the 3D surface, with improved details for the objects of the scene nearest to the camera. By using the same algorithm, we address the traveling salesman problem and large size instances with up to 33708 cities. For SOM applications, a basic characteristic is the many spiral search of closest points in the plane, each one performed in time complexity $O(1)$, in average when dealing with a bounded data distribution. Then, one of the main interests of the proposed approach is to allow the execution of approximately N spiral searches in parallel, where N is the problem size. This is what we call “massive parallelism”, the theoretical possibility to reduce average computation time by factor N, and many repetitions of constant time simple operations. We systematically study the influence of problem size, together with the trade-off between solution quality and computation time on both CPU and GPU, in order to gauge the benefit of massive parallelism.

1.3/ PLAN OF THE DOCUMENT

According to the objectives and concerns, this thesis is organized into two main parts: one part is related to background definitions and exposition of state-of-the-art methods, the other part is devoted to the proposed model, solution approaches and experiments. The plan is summarized in Fig. 1.1.

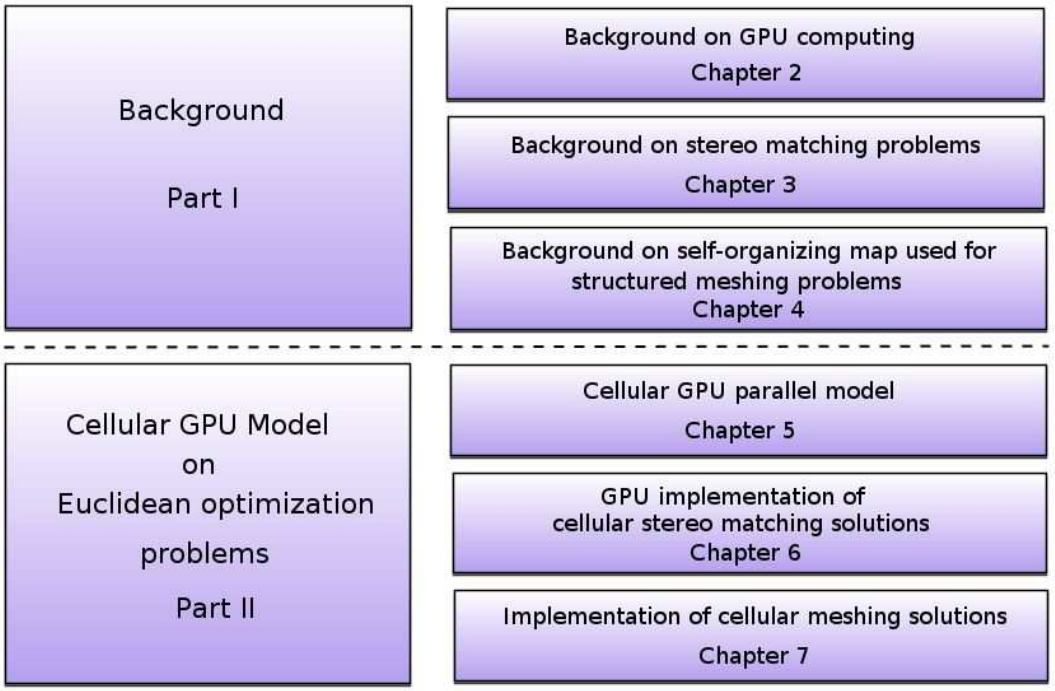


Figure 1.1: Reading directions

The first part is composed of chapters 2, 3, and 4. Chapter 2 presents the GPU computation architecture and its related CUDA programming model. Chapter 3 presents background on stereovision, and reviews local dense stereo-matching method already applied on CPU and GPU plateforms. Chapter 4 presents background on the self-organizing map and of its application on structured meshing and transportation problems. For each application and algorithm, these chapters try to present the state-of-the-art in parallel GPU computing for such problems.

The second part of the document is composed of chapters 5, 6, and 7. Chapter 5 presents the cellular GPU parallel model for massive parallel computation for Euclidean optimization problems. Chapter 6 presents application to stereo-matching problems for CFA demosaicing stereovision and the real-time stereo-matching implementation that is derived from the standard local dense scheme. Chapter 7 presents details about the two GPU applications of the cellular parallel SOM algorithm. The two applications are the balanced structured mesh problem applied on disparity map, and the well-known Euclidean traveling salesman optimization problem.

Then, a general conclusion finishes the document. A section specially exposes the perspectives of this work for parallel optimization computation in the future.

I

BACKGROUND

2

BACKGROUND ON GPU COMPUTING

2.1/ INTRODUCTION

Most personal computers can now integrate GPU cards at very low cost. That is the reason why it would be very interesting to exploit this enormous capability of computing to implement parallel models for optimization applications. In this chapter, we will introduce the background of the GPU architecture and the CUDA programming environment used in our work. Specifically, we will first present the GPU hardware organization and memory system. Then we will give general introductions to the CUDA programming model and the Single-Program Multiple-Data (SPMD) parallel programming paradigm, explain the thread and memory management, outline the scalability and synchronization effects of this model, and introduce the configuration of block and grid in multi-thread execution.

2.2/ GPU ARCHITECTURE

In this section, we provide a brief background on the GPU architecture. Our analytical model is based on the Compute Unified Device Architecture (CUDA) programming model and the NVIDIA Fermi architecture [GTX13] used in the GeForce GTX 570 GPU.

2.2.1/ HARDWARE ORGANIZATION

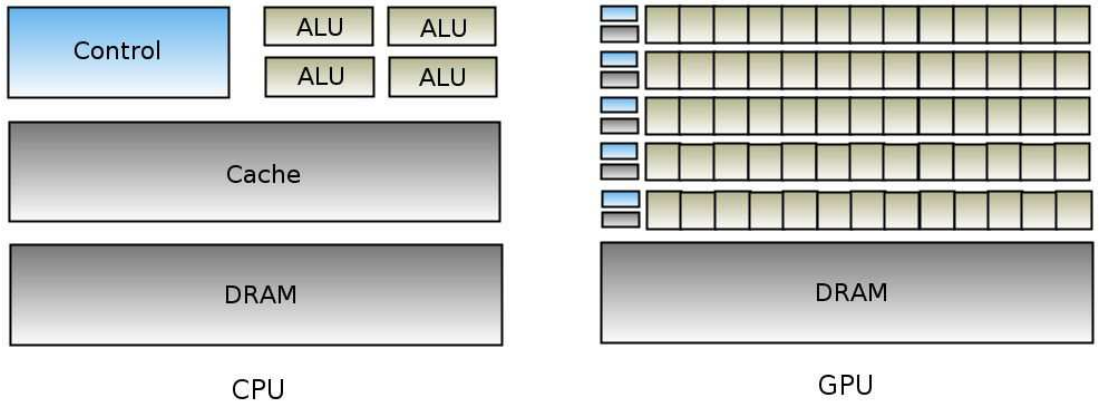


Figure 2.1: Repartition of transistors for CPU and GPU architectures.

For years, the use of graphics processors was dedicated to graphics applications. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multi-threaded and many-core environment. Indeed, this architecture provides a tremendous powerful computational capability and a very high memory bandwidth compared to traditional CPUs. The repartition of transistors between the two architectures can be illustrated as Fig. 2.1.

We can see in Fig. 2.1 that CPU does not have a lot of Arithmetic-Logic Units (ALU in the figure), but a large cache and an important control unit. And therefore, CPU is specialized for management of multiple and different tasks in parallel that require lots of data. In this case, data are stored within a cache to accelerate its accesses. While the control unit will handle the instructions flow to maximize the occupation of ALU, and to optimize the cache management. In other hand, GPU has a large number of arithmetic units with limited cache and few control units. This architecture allows the GPU to compute in a massive and parallel way the rendering of small and independent elements, while having a large flow of data processed. Since in GPU, more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for intensive and highly parallel computations.

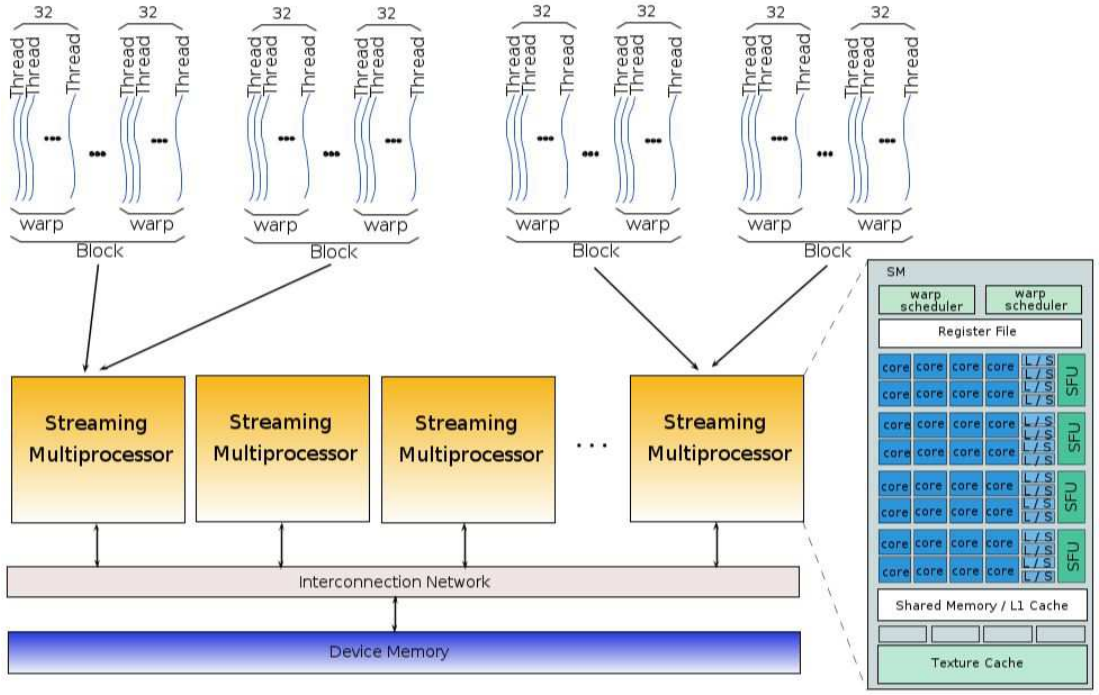


Figure 2.2: A sketch map of GPU architecture.

Fig. 2.2 provides a display of the global GPU architecture. The GPU architecture consists of a scalable number of streaming multiprocessors (SMs). For GTX 570, the number of SM is 15, and each SM features 32 single-precision streaming processors (SPs), which are more usually called CUDA cores, four special function units (SFUs) executing transcendental instructions such as sin, cos, reciprocal and square root. Each SFU executes one instruction per thread per clock and a warp executes over eight clocks. A 64KB high speed on-chip memory (Shared Memory/L1 Cache) and an interface to a second cache are also equipped for each SM [Gla13].

The SM executes a set of 32 threads together called a warp. Executing a warp instruc-

tion consists of applying the instruction to 32 threads. In the Fermi architecture a warp is formed with a batch of 32 threads. The GPU of the Fermi architecture uses a two-level, distributed thread scheduler for warp scheduling. The *GigaThread Engine* is above the SM level and the *Dual warp Scheduler* at the SM level, the later is the one more usually concerned. Each SM can issue instructions consuming any two of the four blue execution columns shown in Fig. 2.2. For example, the SM can mix 16 operations from the 16 cores in the first two-column with 16 operations from the 16 cores in the second two-column or 16 operations from the load/store units or any other combinations the program specifies. Normally speaking, one SM can issue up to 32 single-precision (32-bit) floating point operations or 16 double-precision (64-bit) floating point operations at a time. More precisely, at the SM level, each warp scheduler distributes warps of 32 threads to its execution units. Threads are scheduled in warp. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The dual warp scheduler selects two warps and issues one instruction from each warp to a group of 16 cores, 16 load/store (shown as the L/S units in Fig. 2.2) units or 4 SFUs. Most instructions can be dually issued, such as two integer instructions, two floating instructions or a mix of integer, floating point. Load, store and SFU instructions can be issued concurrently also. Double precision instructions do not support dual dispatch with any other operations.

2.2.2/ MEMORY SYSTEM

2.2.2.1/ MEMORY HIERARCHY

From a hardware point of view, GPU consists of streaming multiprocessors, each with processing units, registers and on-chip memory. As multiprocessors are organized according to the SPMD model, threads share the same code and have access to different memory space. Table 2.1 and Fig. 2.3 show these different available memory space and connections with threads and blocks.

Table 2.1: GPU MEMORY SPACE.

Memory Type	Access Latency	Size
Global	Medium	Large
Registers	Very fast	Very small
Local	Medium	Medium
Shared	Fast	Small
Constant	Fast (cached)	Medium
Texture	Fast (cached)	Medium

The communication between CPU and GPU is done through the global memory. However, in most GPU configurations, this memory is not cached and its access is quite slow, people have to minimize accesses to global memory for both read/write operations and reuse data within the local multiprocessor memory space. GPU has also read-only texture memory to accelerate operations such as 2D and 3D mapping. Texture memory space can be used for fast graphic operations. It is usually used by binding texture on global memory. Indeed, it can improve random accesses or uncoalesced memory access patterns that occur in common applications. Constant memory is another read-only

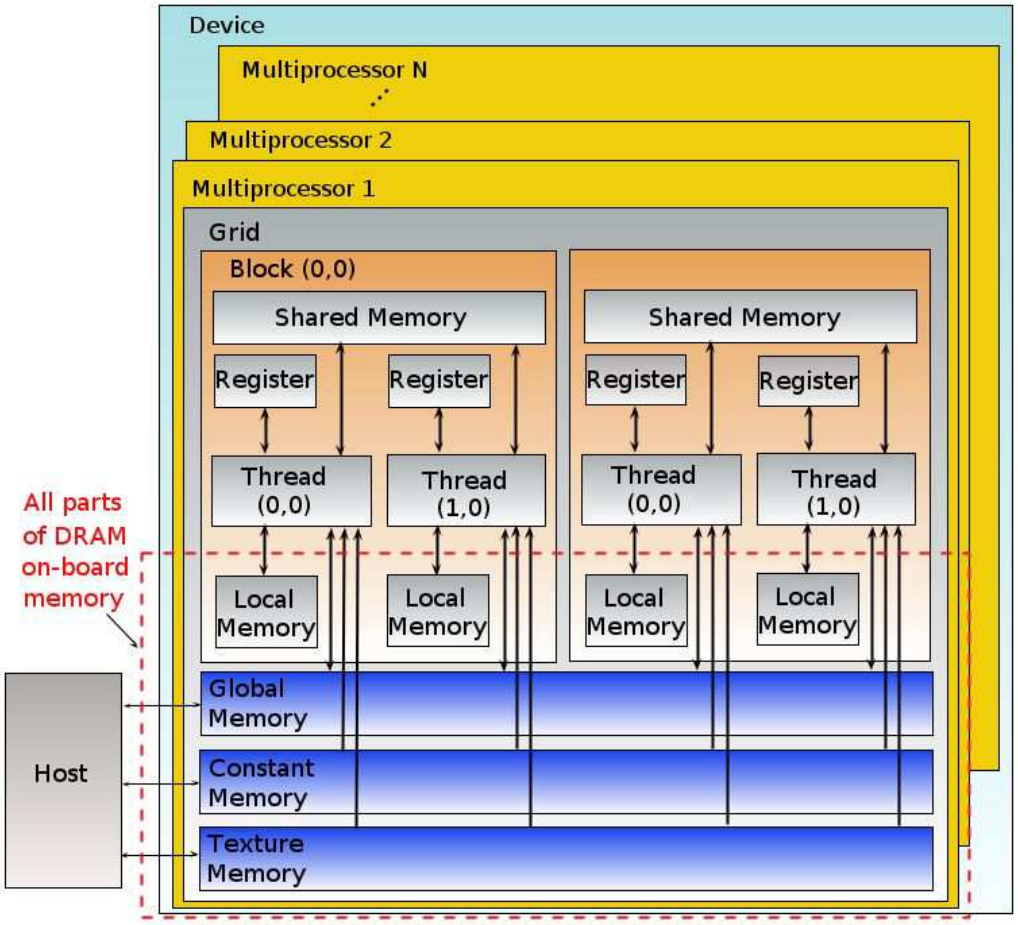


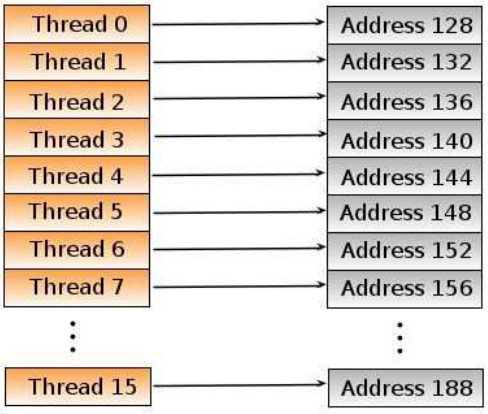
Figure 2.3: CUDA Programming Model.

memory, but its hardware is optimized for the case where all threads read the same location. Shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. This memory area provides a way for threads to communicate within the same block. Registers among streaming processors are exclusive to an individual thread; they constitute a fast access memory. In a kernel code, that is a global CUDA funtion, each declared variable is automatically put into registers. Local memory is a memory abstraction and is not an actual hardware component. In fact, local memory resides in the global memory allocated by the compiler. Complex structures such as declared arrays will reside in local memory. In addition, the local memory is meant as a memory location used to hold "spilled" registers. Register spilling occurs when a thread block requires more registers than available ones on an SM. Local memory is used only for some automatic variables, which are declared in the device code without any of the qualifiers, such as __device__, __shared__, or __constant__. Generally, an automatic variable resides in a register except for the following cases: arrays that the compiler cannot determine are indexed with constant quantities and large structures or arrays that would consume too much register space. Any variable can be spilled by the compiler to local memory, when a kernel uses more registers than that are available on the SM.

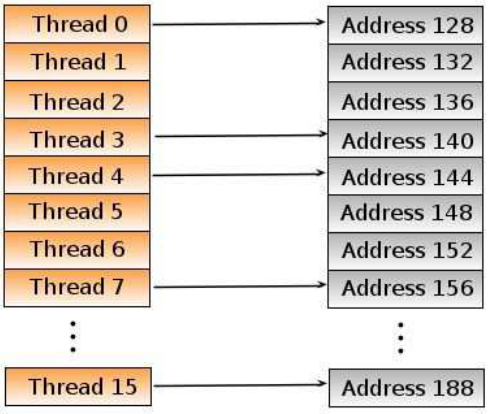
2.2.2.2/ COALESCED AND UNCOALESCED GLOBAL MEMORY ACCESSES

Regarding the executing processing, the SM processor executes one warp at one time and schedules warps in a time-sharing fashion. The processor has enough functional units and register read/write ports to execute 32 threads together. When the SM processor executes a memory instruction[1], it generates memory requests and switches to another warp until all the memory values in the warp are ready. Ideally, all the memory accesses within a warp can be combined into one memory transaction. In fact, for best performance, accesses by threads in a warp must be coalesced into a single memory transaction of 32, 64 or 128 bytes. Unfortunately, that depends on the memory access pattern within a warp. If the memory addresses are sequential, all of the memory requests within a warp can be coalesced into a single memory transaction. Otherwise, each memory address will generate a different transaction. Fig. 2.4 illustrates two examples for each of these two cases. The CUDA manual [NVI10] provides detailed algorithms to identify types of coalesced/uncoalesced memory accesses. If memory requests in a warp are uncoalesced, the warp cannot be executed until all the memory transactions from the same warp are serviced, which will take significantly longer time than waiting for only one memory request as in the coalesced case and it can lead to a significantly performance decrease. However, some modifications have been done as a solution to the problem of latency in the case of uncoalesced memory accesses. Generally speaking, before the GPU compute capability version 1.3, stricter rules are applied to be coalesced. When memory requests are uncoalesced, one warp generates 32 memory transactions. While in the later versions after version 1.3, the rules are more relaxed and all memory requests are coalesced into as few memory transactions as possible [HK09].

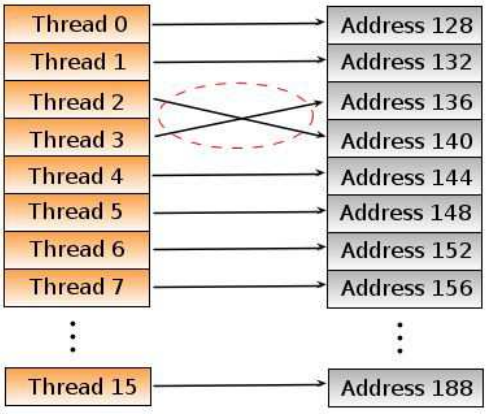
[1] In this document, a computation instruction refers to a non-memory instruction



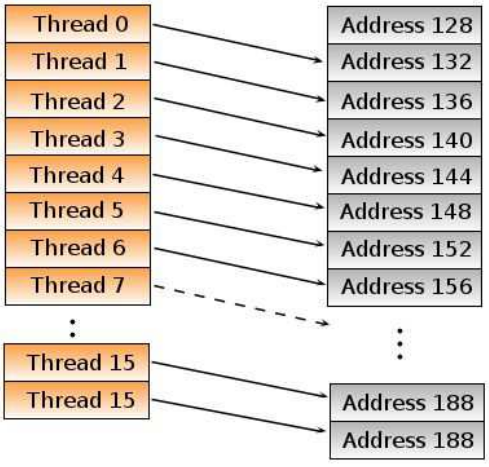
(a) **Example 1_1**: Coalesced **float** memory access resulting in single memory transaction.



(b) **Example 1_2**: Coalesced **float** memory access (divergent warp) resulting in single memory transaction.



(c) **Example 2_1**: Non-sequential **float** memory access resulting in sixteen memory transactions.



(d) **Example 2_2**: Access with misaligned starting address **float**, resulting in sixteen memory transactions.

Figure 2.4: Coalescence and uncoalescence phenomenon in global memory access.

2.2.2.3/ On-chip shared memory

While the global memory is part of the off-chip Dynamic Random Access Memory (DRAM), the shared memory is implemented within each SM multiprocessor as a Static Random Access Memory (SRAM). The shared memory has very low access latency, which is almost the same as that of register, normally 10-20 cycle, and high bandwidth of 1,600 GB/s, which has been widely investigated to reduce non-coalesced global memory accesses in regular applications [SRS+08]. However, since one warp of 32 threads accesses the shared memory together, when there is a bank conflict within a warp, accessing the shared memory takes multiple cycles. Moreover, this on-chip memory (Shared Memory/L1 Cache) can be used either to cache data for individual threads (as register spilling/L1 Cache) and/or to share data among several threads (as shared memory). This 64 KB memory can be configured as either 48 KB of shared memory with 16 KB of L1 cache, or 16 KB shared memory with 48 KB of L1 cache. Shared memory enables threads within the same thread block to cooperate, facilitates extensive reuse of on-chip data, and greatly reduces off-chip traffic. Shared memory is accessible by the threads in the same block.

2.3/ CUDA

2.3.1/ The CUDA programming model

CUDA is an acronym standing for Compute Unified Device Architecture. It is a parallel computing platform and programming model created by NVIDIA and implemented in their graphics processing units (GPUs). CUDA provides the access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Thanks to CUDA, the latest NVIDIA GPUs become accessible for computation like CPUs. GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly rather than executing one single thread very quickly. The arrival of CUDA leverages a powerful parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on CPUs. Normally, the approach of solving general-purpose (not exclusively graphics) problems on GPUs with CUDA is known as GPGPU.

CUDA gives developers a software environment that allows using C as a high-level programming language. As illustrated in Fig. 2.5, other languages or application programming interface are supported, such as CUDA FORTRAN, OpenCL and DirectCompute.

When CUDA is executed on GPUs, all the threads will be grouped into blocks and then into warps. All the threads in one block are executed on one SM together. One SM can also have multiple concurrently running blocks. The number of blocks, which are running on one SM, is determined by the resource requirements of each block, such as the number of registers and shared memory usage. The blocks, which are running on one SM at a given time, are called *active blocks* in this paper. Since typically one block has several warps and the number of warps is the same as the number of threads in one block divided by 32, the total number of active warps per SM is equal to the number of warps per block times the number of active blocks.

Generally speaking, the scheduling of warps is realized by SMs automatically and sequentially. We take a block of 128 threads as an example. The threads in this block will

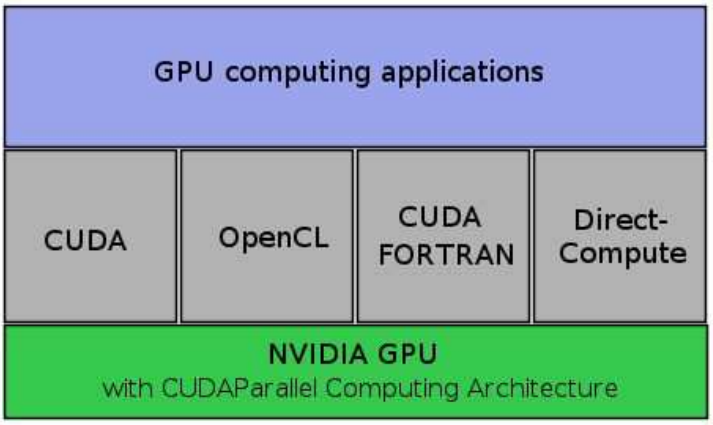


Figure 2.5: CUDA is Designed to Support Various languages or Application Programming Interfaces.

be grouped into four warps: 0-31 threads may be in Warp 1, 32-63 threads may be in Warp 2, 64-95 threads Warp 3 and 96-127 threads Warp 4. However, if the number of threads in the block is not a multiple of 32, the SM (the warp scheduler) will take all the threads left as the last warp. For example, if the number of threads in the block is 66, then there will be three warps: Warp 1 contains 0-31 threads, Warp 2 contains 32-63 threads and Warp 3 takes 64-65 threads. Since the last warp has only two threads, it leads to the waste of computation capability of 30 threads.

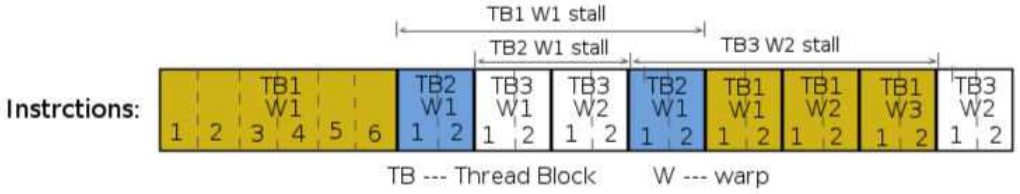


Figure 2.6: Warp execution on SM. The waiting warps will be replaced by the ready warps to hide latency.

At any moment, one SM executes only one warp from one block. But this does not imply that SM will certainly finish all the instructions in this warp all at once. If the executing warp needs to wait for some more information or data (for example: reading from/writing in the global memory), a second warp will be shifted into the SM to replace the executing warp for the purpose of hiding latency, as shown in Fig. 2.6. So there should be one theoretically best situation for performance: all the SMs have enough warps to shift if it is necessary, and all the SMs are busy in the execution duration. It is one of the key points to obtain high performance. And it is the reason why, it is necessary to use the threads and blocks in a way that maximizes hardware utilization. In other words, best performance can be reached when the latency of each warp is completely hidden by other warps [OHL+08, JD10]. To achieve this, a GPU application can be tuned by two leading parameters: the number of threads per block and the total number of threads.

2.3.2/ SPMD PARALLEL PROGRAMMING PARADIGM

The Single-Program Multiple-Data (SPMD) paradigm is the most used paradigm in parallel algorithm designing. In this paradigm, processors execute basically the same piece of code but on different parts of the data, which involves the splitting of application data among the available processors. In some other papers, this paradigm is also referred to as geometric parallelism, domain decomposition, or data parallelism. A schematic representation of this paradigm can be seen in Fig. 2.7

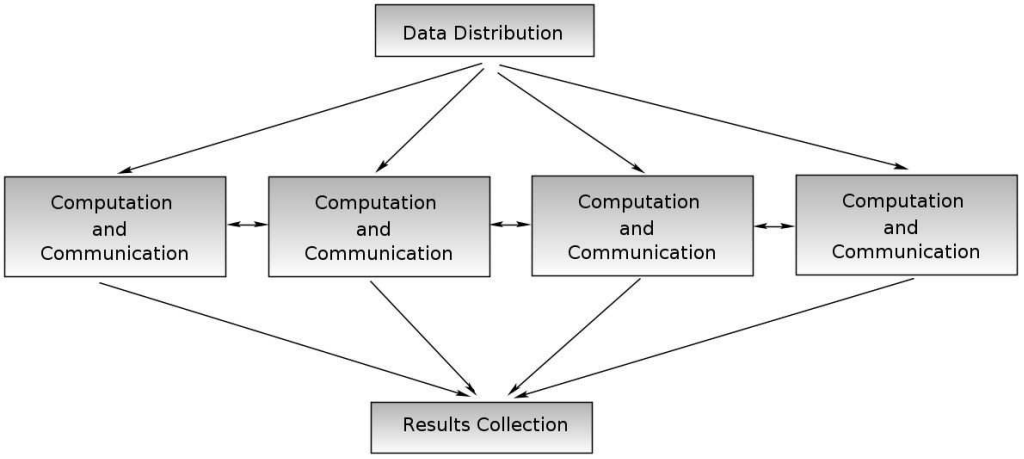


Figure 2.7: Basic processing structure of SPMD paradigm.

The applications in this document all have an underlying regular geometric structure, such as the image plane with pixels in stereo matching applications, structured meshing problem, and city distribution in plane in TSP. This allows the data to be distributed among the processors following a given way, where each processor will be in charge of a defined spatial area. Processors communicate with neighboring processors and the communication load will be proportional to the size of the boundary of the input element, while the computation load will be proportional to the volume of the input element. In certain platforms, it may also be required to perform some global synchronization periodically among all the processors. The communication pattern is usually highly structured and extremely predictable. The data may initially be self-generated by each process or may be read from the main memory space during the initialization stage.

SPMD applications can be very efficient if the data are well distributed among the processors and the system is homogeneous. If different processors present different workloads or capabilities, then the paradigm should require the support of some load-balancing scheme able to adapt the data distribution layout during run-time execution [SB08].

It should be noted that this paradigm is highly sensitive to the loss of some processors. Usually, the loss of a single processor is enough to cause a deadlock in the computation processing, in which none of the processors can reach the global synchronization point if it exists.

2.3.3/ THREADS IN CUDA

The CUDA programming model follows SPMD software model. The GPU is treated as a coprocessor that executes data-parallel kernel functions. All CUDA threads are organized into a two level concepts: CUDA grid and CUDA block. A kernel has one grid which contains multiple blocks. Every block is formed of multiple threads. The dimension of grid and block can be one-dimension, two-dimension or three-dimension. Each thread has a *threadId* and a *blockId*, which are built-in variables defined by the CUDA runtime to help user locate the thread's position in its block, as well as its block's position in the grid [NVI12a, SK10].

CUDA provides three key abstractions: one hierarchy of thread groups, device memories and barrier synchronization. Threads have a three-level hierarchy. One block is composed of tens of or hundreds of threads. Threads within one block can share data using shared memory and can be synchronized at a barrier. All threads within a block are executed concurrently on a multi-threaded architecture. A grid is a set of thread blocks that executes a kernel function. Each grid consists of blocks of threads. Then, programmers specify the number of threads per block and the number of blocks per grid.

2.3.4/ SCALABILITY

The advent of GPUs, whose parallelism continues to scale with Moore's law, bring parallel systems into real applications. The challenge left is to develop computational application software, which transparently scales its parallelism to leverage the increasing number of processor cores, such as 3D graphics applications transparently scale their parallelism to GPUs with widely varying numbers of cores.

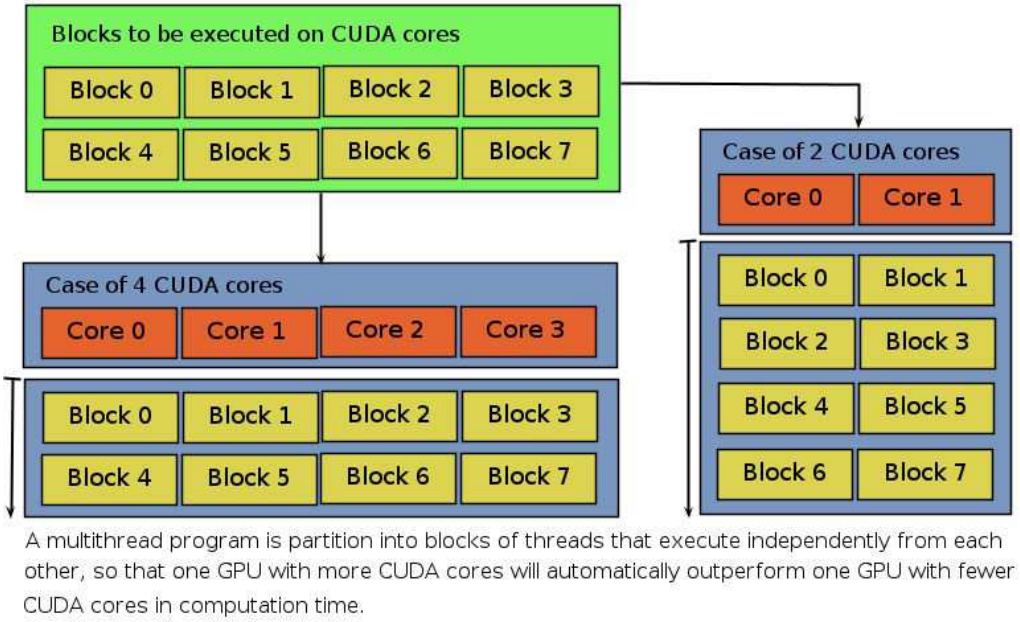


Figure 2.8: Scalable programming model.

The CUDA parallel programming model is designed to overcome this challenge, while maintaining a low learning curve for programmers familiar with standard programming

languages, such as C or C++. In fact, the CUDA is simply exposed to the programmers as a minimal set of language extensions of C, even with the three key abstractions of its core: a hierarchy of thread groups, special memory spaces and barrier synchronization.

These abstractions provide very fine-gained data and thread parallelism. They guide programmers to partition their problems into coarse sub-problems, that can be solved independently in parallel by blocks of threads, and further each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the same block. This decomposition preserves language expressivity, by allowing threads to cooperate when solving each sub-problem, and at the same time, enabling automatic scalability. As it is illustrated in Fig. 2.8, only the runtime system needs to know the count of physical processors. While each block of threads can be scheduled on any of the available processor cores in any order, a compiled CUDA program can be executed on any number of processor cores.

2.3.5/ SYNCHRONIZATION EFFECTS

The CUDA programming model supports thread synchronization through the __syncthreads() function. Typically, all the threads are executed asynchronously whenever all the source operands in a warp are ready. However, if we take this function into count, it will stand as a barrier synchronization function, that makes the threads in the same block coordinate their activities, which will guarantee that all the simultaneously activated threads are in the same location of the program sequence at the same time, thus ensuring that all the threads are reading the correct values from the relative memory space.

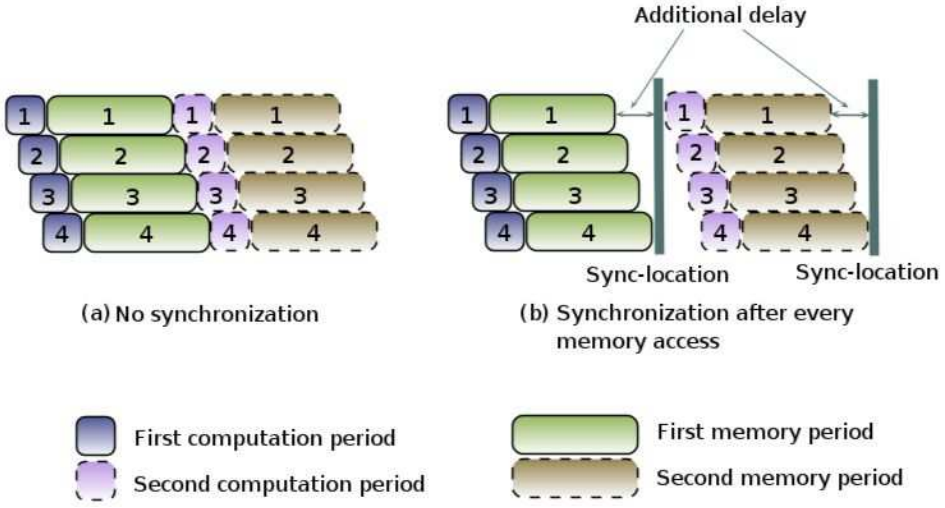


Figure 2.9: Additional delay effects of thread synchronization.

When a kernel calls __syncthreads() function, threads of the same block will be stopped until all the threads of the block reach the sync-location[1]. While this may seem to slow down execution since threads will be idle if they reach the sync-location before other threads, it is absolutely necessary to sync the threads here, because the additional delay

[1]It should be noticed that CUDA has no barrier for synchronize the blocks of a grid, thus blocks can execute in any order relative to each other

is surprisingly less than one waiting period, in almost all the applications [NVI10]. Fig. 2.9 shows the additional delay effect.

2.3.6/ WARP DIVERGENCE

An undesirable effect of having data dependent branching in flow control instruction (if, for, while) is warp divergence. This can slow down the instruction throughput when threads of the same warp follow different execution paths. In that case, the different execution flows in a warp are serialized. Since threads of a same warp share a program counter, this increases the number of instructions executed for this warp. Once all executed paths are completed, the warp converges back to the same execution path. Full efficiency arises when all 32 threads of a warp follow a common path. But these conditions look difficult to obtain in data dependent applications with non-uniform distributions.

2.3.7/ EXECUTION CONFIGURATION OF BLOCKS AND GRIDS

The dimension (both the width and the height) and size (total number of elements) of a grid and the dimension and size of a block are both important factors. The multidimensional aspect of these parameters allows easier mapping of multidimensional problems to CUDA GPU even if it does not play a role in performance. As a result, it is more interesting to take the 'size' into discussion rather than the 'dimensions '.

The number of active warps per multiprocessor has a great influence on the latency hiding and occupancy. This number is implicitly determined by the execution parameters along with resource constraints, such as registers or/and other memory space on chip. The choosing of execution parameters maintains a balance between the latency hiding, occupancy and the resource utilization. There do exist some certain heuristics that can be individually applied to each parameter [NVI11]. When choosing the first execution configuration parameter, the number of blocks per grid, the primary concern is keeping the entire GPU busy. It is better to make the grid size larger than the number of multiprocessors, so that all multiprocessors have at least one block to execute. Moreover, there should be multiple active blocks per multiprocessor, so that blocks that are not waiting for a synchronization function (__syncthreads()) can keep the hardware busy. This recommendation is subject to resource availability, so, it should be determined along with a second execution parameter, the number of threads per block and the usage of memory-on-chip, such as shared memory.

When choosing the block size, it is important to keep in mind one phenomenon, that multiple concurrent blocks can reside on a multiprocessor and therefore occupancy is not determined only by block size alone. In other words, a larger block size does not necessarily lead to a higher occupancy. For example, on a device of computation capacity 1.1 or lower, a kernel with a maximum block size of 512 threads results in an occupancy of 66 percent, because the maximum number of threads per multiprocessor on such a device is 768. Hence, only a single block can be active per multiprocessor. However, a kernel with 256 threads per block on such a device can result in 100 percent occupancy with three resident active blocks [NVI10].

However, higher occupancy does not always mean better performance. A lower occupancy kernel will evidently have more registers available per thread than a higher occu-

pancy kernel, which may result in less register spilling to local memory. Typically once an occupancy of 50 percent has been reached, additional increases in occupancy do not necessarily translate into improved performance [Gup12].

With respect to such factors involved in selecting block size, inevitably trial configurations are required. Given the knowledge of occupancy, there still exist a few rules of thumb to help us set the block size for a better performance:

- Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
- A minimum of 64 threads per block should be used, but this should happen only if there are multiple concurrent blocks per multiprocessor.
- Between 128 and 256 threads per block can be a better choice than others and a good initial range for experimentation with different block sizes.
- It can be very helpful to use some smaller thread blocks rather than one large thread block per multiprocessor if the performance is too much affected by latency, especially when the kernels frequently call the synchronization function (__syncthreads()).

2.4/ CONCLUSION

In this chapter, we have briefly presented the GPU's parallel architecture and the CUDA programming environment. We first presented the GPU architecture, lying out the two cases of memory accesses: the coalesced access and the uncoalesced access to memory space, figuring out that it is more suggested to reach the coalesced access for efficient data read/write in the main memory space. Then, we explained the CUDA programming model, with the presentation of the scalable programming model and the introduction of configuration of blocks and grids in multi-thread execution. At last, we have drawn out the basic frame of the parallel programming platform. In next two chapters, we will present the background of the problems and applications addressed in this document.

3

BACKGROUND ON STEREO-MATCHING PROBLEMS

3.1/ INTRODUCTION

This chapter is centered at the presentation of the background of the stereo-matching problems. We first give a definition of stereo-matching problem, lying out its geometry model and the important definitions used in the document. After that, is the presentation of the stereo-matching method, on which we focus at the local matching methods including their mostly used matching costs and their update in color stereo-matching problems. The left-right consistency check is mentioned as the common used outlier detector for removing the matching errors in the estimated disparity maps. Then, we provide general introductions to the two applications studied in this document: the CFA stereo-matching problem and the real-time stereo-matching implementation. Finally, we enumerate the current research progress in stereo-matching methods.

3.2/ DEFINITION OF STEREOVISION

In the field of automotive vehicles or robot system, main recent applications require the perception of the three-dimensional real world. In this case, the intelligent vehicle system, used in assisting the driver and warning him when there is a potential danger, should be able to detect the different objects that are on the road, and represent them in the three-dimensional scene map. Here, we focus on artificial devices such as stereo cameras, equipped with two cameras, to mimic and simulate the mainly used detecting system, that is, the human-eyes bionic stereovision system. Two images of the scene are acquired simultaneously by the cameras. From these two images (left image and right image), stereovision system aims to recover the third dimension, which has been lost during the image formation. The projections of the same scene point visible by the two cameras do not have the same coordinates in the two image planes. The exact position of this scene point can be retrieved if its two projections, called homologous pixels, are identified. The problem of identifying homologous pixels in the two images is called stereo-matching problem.

Stereo matching methods are applied to pairs of stereo images that can be gray-level images or color images. In gray-level images each pixel is characterized by a gray-level intensity value, while in color images each pixel is characterized by three color components Red (R), Green (G) and Blue (B) intensities.

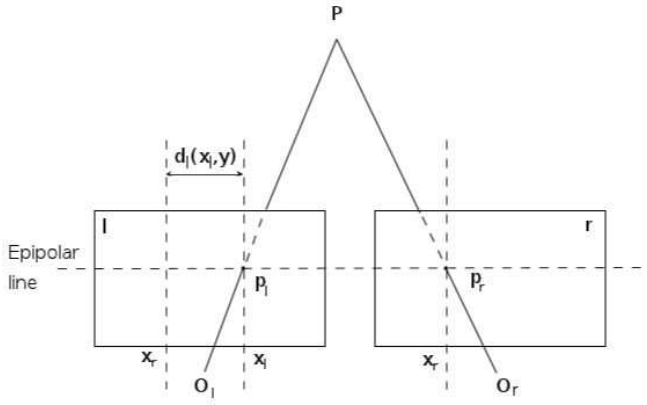


Figure 3.1: Binocular Stereovision Geometry.

In a classic binocular stereoscopic vision, from a scene point *P*, using the projection matrices of the left and the right cameras, we can find the location of the projected points onto the left and the right image plans, as illustrated in Fig. 3.1. There is a very interesting set of properties, called epipolar geometry, related to the classic binocular stereovision. Several terms are defined as following:

- the **epipolar plane** of a scene point *P* is the plane determined by *P* and the projection centers O_l and O_r.

- the **left epipolar** is the projection onto the left projection plane of the right projection center and the **right epipolar** is the projection onto the right projection plane of the left projection center.

- to any space point, we associate two epipolar lines which are in overlap as one line in Fig. 3.1. The epipolar lines are the intersections of the epipolar plane of the point with the projection planes. In the figure, the epipolar line is the projection of the straight line O_lP or O_rP onto the right or the left projection plane, respectively.

The epipolar geometry describes the relation between right and left projections of a scene point *P*. Hence, the very important property, called epipolar property is introduced: **given a scene point *P*, its right projected point p_r lies on the right epipolar line corresponding to its left projected point p_l and vice versa**.

Now suppose an inverted problem, the two projected points are identified and we want to find the location of scene point *P*. *P* is the intersection of the straight lines O_lp_l and O_rp_r. So the scene point *P* can be recovered if only the pair of left and right projected points is identified. Given only one projection of a scene point *P*, the stereo-matching problem aims at determining its homologous one in the other image plan if it exists.

As we know, for one projection *p*, its homologous projection *p'*, if it exists, lies on the epipolar line corresponding to the projection *p*. However, in the binocular model, the two epipolar lines coincide with each other. So, the correspondence problem is in fact a one-dimensional search problem rather than one two-dimensional search problem. However, the homologous point might not even exist in certain cases, such as half-occlusion.

3.2.1/ HALF-OCCLUSION

In binocular stereovision, since there are some scene points that can be visible by only one camera, we cannot project every scene point onto two image plans [OA05]. We refer to these points as half-occluded. The Fig. 3.2 presents this very half-occlusion phenomenon and the Fig. 3.3 shows a more concrete example. In Fig. 3.2, the scene *B* is projected onto b_l in the left image plane but is not visible by the right camera. Similarly, the scene point *C* is projected onto c_r in the right image plane but is not visible by the left camera. We can deduce that all the scene points between *A* and *B* are invisible for the right camera, while the scene points between *C* and *D* are invisible for the left camera.

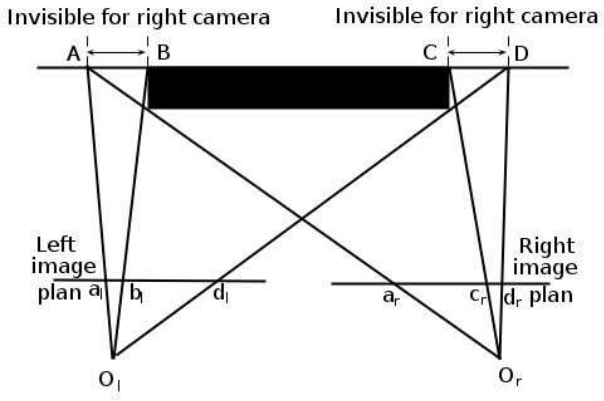


Figure 3.2: Half-occlusion Phenomenon.

(a) Left image acquired by left camera.

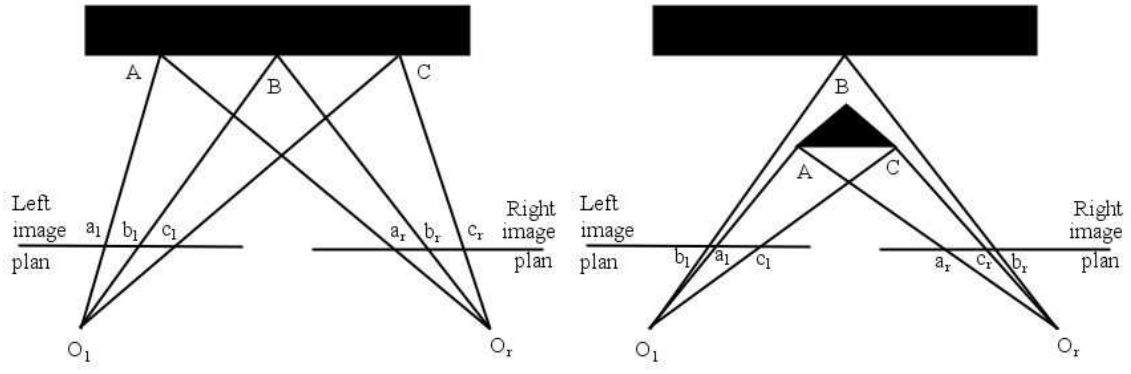
(b) Right image acquired by right camera.

(c) Images representing the different parts of (a) and (b). The four parts from left to right are: visible only by left camera, visible by both cameras, visible by both cameras and visible only by right camera.

Figure 3.3: Half-occlusion phenomenon in example of image pair “Teddy”.

3.2.2/ ORDER PHENOMENON

In the stereovision model, an assumption is usually taken: a group of scene points has the same projection rank order in both image plans. Fig. 3.4(a) presents a case that respects this order. However, this hypothesis of order conservation does not always hold true. For example, when the scene points belong to objects at different distances from cameras, this order is not respected any more, as illustrated in Fig. 3.4(b).



(a) Order Respected. (b) Order Not Respected.

Figure 3.4: Order Phenomenon.

3.3/ STEREO MATCHING METHODS

3.3.1/ METHOD CLASSIFICATION

Generally speaking, a high computational complexity is not avoidable for methods to solve the stereo-matching problem by analyzing a pair of stereo images. Particularly, for each pixel in the left image, there are a lot of possible candidate right pixels to be examined in order to determine the best correspondence. It is assumed that the homologous right pixel corresponds to the best correspondence. Stereo-matching methods can be sorted into two classes: the sparse methods and the dense methods [Wor07].

Sparse methods match features that have been identified in the stereo image pair. The features used can be edges, line segments, curve segments and so on. For this reason, these methods are also called feature-based methods. the matching process is carried out only on the detected features [Wu00]. These methods draw significant attention for many years since 1980s because of their low computational complexity. In that time, they are well suited for real-time applications [MPP06]. However, these methods can not satisfy those applications that need an accurate identification of all the homologous pixels in the stereo image pair [BBh03]. This turns people to dense methods in the last decade.

The dense matching methods are the stereo-matching methods that provide all the homologous pixels in the stereo image pair. The two methods proposed in this document are both dense matching methods. Since there are a large amount of papers about solving the stereo-matching problem, it is quite difficult to make an exhaustive review. But

as these methods do have the same processing steps, a brief review of recent dense stereo-matching methods has been done by Scharstein and Szeliski [SS02].

3.3.2/ DENSE STEREO-MATCHING METHOD

The four steps which are usually performed by a dense stereo-matching method are identified:

- Matching cost computation.
- Cost aggregation where the initial matching costs are spatially aggregated over a support region of a pixel.
- Optimization to determine the best correspondence at each pixel.
- Matching results refinement to remove the outliers.

The dense matching methods are further classified into global methods and local methods. The optimization step of global methods involves a high computational effort which does not make them suitable for real-time applications. Although global methods can provide very good results [SS02, BBh03], the interest in local methods does not decrease thanks to their simplicity and low computational complexity, and the local methods have the top performance [Mei11, SWP+12] in the list of Middlebury [HS06].

In a context of GPU computation, local dense stereo-matching methods, which are also called window-based approaches, are the natural choice for parallel computation. These methods assume that the intensity configurations are similar in the neighborhood of homologous pixels. Particularly, intensity values of neighbors of a pixel in the left image are closed to those of the same neighbors of its homologous pixel in the right image. So, a matching cost is defined between the window around the left pixel and the window around the candidate pixels in the right pixels. The window is also called support region.

3.3.3/ MATCHING COST COMPUTATION

Common pixel-based matching costs include absolute differences (AD), sum of absolute differences (SAD), squared differences (SD), sum of squared differences (SSD) and sampling-insensitive absolute differences [BT98], or their truncated versions. A detailed review about matching costs is provided by [HS07]. Since stereo-matching costs are used by window-based stereo methods, they are usually defined for a given window shape.

For a fixed window in a gray-level image, the Absolute Difference (AD), between the gray-level $I_l(x_l, y)$ of pixel p_l with coordinates (x_l, y) in the left image and the gray-level $I_r(x_l - s, y)$ of a candidate pixel p_r at a shift s with coordinates $(x_l - s, y)$ in the right image, is defined in Equation 3.1, where the subscript g refers to gray-level images. The Sum of Absolute Differences (SAD) is defined as Equation 3.2. The SAD measures the aggregation of absolute differences between the gray-levels of pixels, in support region of size $(2w + 1) \times (2w + 1)$ centered at p_l and a similar window at p_r, with w the window's half-width.

$$AD_g(x_l, y, s) = |I_l(x_l, y) - I_r(x_l - s, y)| \tag{3.1}$$

$$SAD_g(x_l, y, s) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} |I_l(x_l + i, y + j) - I_r(x_l + i - s, y + j)| \tag{3.2}$$

Similarly, the Squared Difference (SD) and the Sum of Squared Differences (SSD) are defined in Equation 3.3 and Equation 3.4, respectively.

$$SD_g(x_l, y, s) = (I_l(x_l, y) - I_r(x_l - s, y))^2 \tag{3.3}$$

$$SSD_g(x_l, y, s) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} (I_l(x_l + i, y + j) - I_r(x_l + i - s, y + j))^2 \tag{3.4}$$

In color images, for each pixel at coordinates (x, y), the presentation of the pixel's dense information is associated with three color components rather than one component in gray-level images. So, to present a point in three-dimensional *RGB* color space, the coordinates of the color point should be updated as $R(x, y), G(x, y)$ and $B(x, y)$. Therefore, a color image can be considered as an array of color points $I(x, y) = (R(x, y), G(x, y), B(x, y))^T$. And the color image can be split into three component images *R*, *G* and *B*, in each of these images, a point is characterized by one single color component level as in gray-level images. A lot of research has shown that the use of color images rather than gray-level ones can highly improve the accuracy of stereo-matching results [Cha05, CTB06, Kos96]. The color information can be sometimes helpful in reducing stereo-matching ambiguities as presented by [CTB06]. Anyway, in most cases, a full color image carries more information in its three color components than a gray-level image of the same scene.

The generalization to color images of stereo-matching costs should also be updated. Based on Equation 3.1 and Equation 3.2, the matching cost AD and SAD are rewritten as Equation 3.5 and Equation 3.6.

$$AD_c(x_l, y, s) = |R_l(x_l, y) - R_r(x_l - s, y)| + |G_l(x_l, y) - G_r(x_l - s, y)| + |B_l(x_l, y) - B_r(x_l - s, y)| \tag{3.5}$$

$$\begin{aligned} SAD_c(x_l, y, s) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} (&|R_l(x_l + i, y + j) - R_r(x_l + i - s, y + j)| \\ &+|G_l(x_l + i, y + j) - G_r(x_l + i - s, y + j)| \\ &+|B_l(x_l + i, y + j) - B_r(x_l + i - s, y + j)|) \end{aligned} \tag{3.6}$$

Similarly, the SD and SSD can be generalized to deal with color images as Equation 3.7 and Equation 3.8. Where the $\|\cdot\|$ is the Euclidean norm, and therefore, $\|\cdot\|^2$ is the squared Euclidean distance between two points of the three-dimensional *RGB* color space. And I_l and I_r are the color points associated respectively with the left and right pixels [Kos93].

$$SD_c(x_l, y, s) = \|I_l(x_l, y) - I_r(x_l - s, y)\|^2 \tag{3.7}$$

$$SSD_c(x_l, y, s) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} \|I_l((x_l + i, y + j) - I_r(x_l + i - s, y + j)\|^2 \tag{3.8}$$

The matching cost will be computed by shifting the window over all the possible candidate pixels in the right image. The final estimated disparity is determined by the shift where the matching cost reaches the minimum. All the local stereo-matching methods use the window in a certain way and they are also called area-based or window-based matching methods.

3.3.4/ WINDOW FOR COST AGGREGATION

Local dense matching methods exploit the concept of support region or window. Every pixel receives a support from its neighbor pixels. It is commonly accepted that pixels inside this support region are likely to have the same disparity and can therefore help to resolve matching ambiguities. Usually, the support region can be classified into fixed window and adaptive window.

A straightforward aggregation approach consists in using a square window centered at each pixel p. This kind of square-window approach implicitly assumes that the disparity is similar over all pixels in square window. However, this assumption does not hold true near discontinuity areas. To overcome this problem, several works have been done [BI99, FRT97], and the shifting window approach is proposed for this purpose. This approach considers multiple square windows centered at different locations and retains the window with the smallest cost. The size of the support window is fixed and is difficult to adjust the size for both square-window and shifting-window approaches. A small window may not include enough intensity values for a good matching result, while a large one may violate the assumption of constant disparity inside the support window. In fact, the window size should be able to represent the shape of objects in one image. Thus, the window size should be large for textureless regions and small for well textured regions. For this reason, Kanade *et al* [KO94] proposes an adaptive-window method which automatically selects the window size and/or shape based on local information. We will retain this method for improving our GPU method.

3.3.5/ BEST CORRESPONDENCE DECISION

After the computation and aggregation of matching cost, the homologous pixel in the right image is derived based on the Winner-Takes-All (WTA) principle, as illustrated in Fig. 3.5. The shift for which the matching cost is the lowest is selected. Thus, the estimated disparity $\hat{d}_l^w(x_l, y)$ at the pixel p_l corresponds to the shift s of the right pixel at which the matching cost is the minimum. It can be expressed as Equation 3.9 when the SAD matching cost is used.

$$\hat{d}_l^w(x_l, y) = arg\min_s(SAD_g^w(x_l, y, s)) \tag{3.9}$$

In this document, the left image pixel is used as the reference in the cost computation as default. So, we use the *l* subscript in the estimated disparity symbol, *s* ranging from s_{min} to s_{max}. And the superscript *w* is taken into use when the aggregation is based on a neighborhood window with a half-width *w*.

Once the disparity has been estimated at each pixel in the left image, the left dense estimated disparity map is formed. The disparity map is the array of disparity values

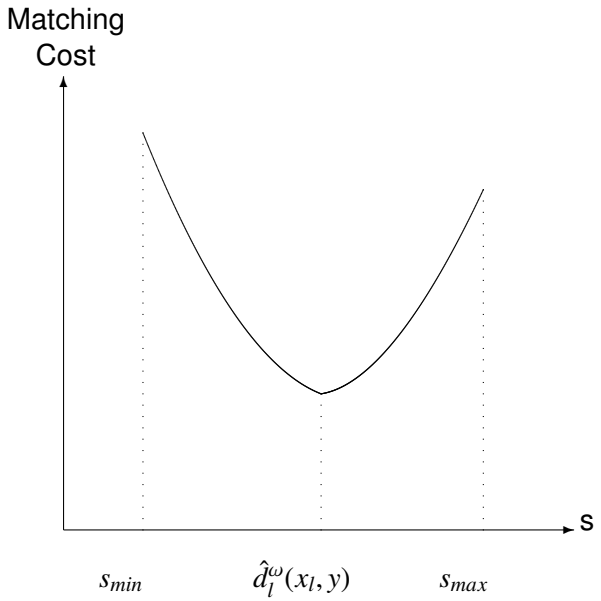


Figure 3.5: Winner-Takes-All method, $[s_{min}, s_{max}]$ represents the possible shifts of the searched pixels. When the shifts s equals to $\hat{d}$ (estimated disparity), the matching cost value reaches the minimum.

computed for each pixel, which has the same size as the input left image and right image.

However, the matching cost does not always reach a global extremum at the correct disparity for all the regions, Fig. 3.6 illustrates three examples of untextured area, textureless area and repetitive area [Cha05], respectively. In these cases, the gray-levels of the left image and right image are represented. If we examine the gray-level of nine pixels, we are not able to determine a correspondence in the right image, because the gray-level pattern is the same along the correspondence epipolar lines in the two images. As a result, several minimums are found, which will shield the true disparity from being chosen. To avoid this problem, the matching cost and the aggregation support region should be carefully chosen to correctly match pixels.

3.3.6/ LEFT-RIGHT CONSISTENCY CHECK

In the matching process, one image is taken as a reference, for each pixel in this image, we seek its homologous pixel in the other image. The matching method may yield one estimated disparity map for each of the two input images. The first one is for the left input image and the second one is based on the right image. On these two estimated disparity maps, there could be some matching errors. So, a refinement step is usually employed to improve the matching quality.

Different methods allow the refinement to improve the disparity estimation quality. These refinement methods are variously used in most stereo-matching methods [FHZF93]. And they are employed as a post-processing step to improve disparity maps by removing those false matching results or for providing sub-pixel disparity estimation.

The most used method for detecting false matching results is the left-right consistency check. This consistency check method takes the estimated disparity as outlier if Equation

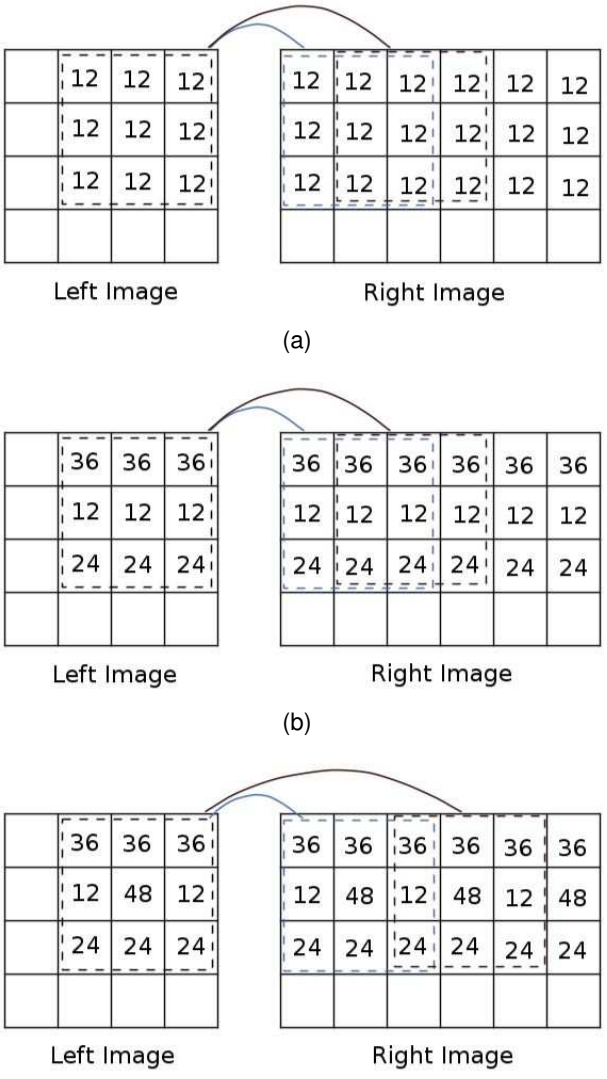


Figure 3.6: Matching cost reaches several minimum. (a) Untextured regions. (b) Textureless regions. (c) Repetitive texture regions.

3.10 does not hold true.

$$\hat{d}_l^w(x_l, y) = \hat{d}_r(x_r, y) = \hat{d}_r(x_l - \hat{d}_l(x_l, y), y) \tag{3.10}$$

If this condition is not verified, then it is considered that this pixel may be bad matched and should be repaired, or lying in a half-occlusion region, and so, no disparity value can be estimated at this pixel [FRT00].

3.4/ TWO IMPLEMENTATIONS OF STEREO-MATCHING PROBLEMS

In this document, two problems of stereo-matching and their related methods are studied and implemented with GPU computing. They are the CFA stereo-matching problem and the real-time stereo-matching. Here, we briefly present their main characteristics.

3.4.1/ INTRODUCTION TO CFA STEREO-MATCHING PROBLEMS

In most real-time implementation of stereo-matching, such as the intelligent cars and the integrated robot systems, the color image pairs can be acquired by two types of cameras: the one equipped with three sensors associated with beam splitters and color filters, providing the so-called full color images, of which each pixel is characterized in Red, Green and Blue levels, and the one equipped only with a single-sensor.

In the second case, the single-sensor cannot provide a full color image directly, but actually deliver a color filter array (CFA) image. Every pixel in it is characterized by a single color component, that can be one of the three color components: Red (R), Green (G) and Blue (B). So, the missing color components have to be estimated at each pixel. This process of estimating the missing color components is usually referred as CFA demosaicing. It produces a demosaiced color image where every pixel is presented by an estimated color point [BGMT08].

As the demosaicing methods intend to produce demosaiced color images, they attempt to reduce the presence of color artifacts, such as the false colors or zipper effects, by filtering the images [YLD07]. So some useful color information for stereo-matching may be lost in the color demosaiced images. As a result, the demosaiced color image pairs' stereo-matching quality usually suffers either from color artifacts or from the alteration of color texture caused by demosaicing schemes.

The method that is presented by Halawana [Hal10] is an alternative solution to match pixels by analyzing directly the CFA images, without reconstructing all the full color image by demosaicing processing. These type of stereovision is called CFA stereo-matching problem. We present and study a GPU implementation of the Halawana method in this document.

3.4.2/ INTRODUCTION TO REAL-TIME STEREO-MATCHING METHODS

The constraint of computation time in stereo-matching methods is very important for applications that run at video rate. So, the running time of stereo-matching methods used

should be less than 34 ms, which is generally the image acquisition time in filming the videos. The research work about real-time stereo-matching follows two main ways:

- looking for algorithms with low computation time while still providing good matching quality,
- developing hardware devices for efficient implementation in order to decrease the computation time.

These devices include special purpose hardware, such as the digital signal processors (DSP) or field programmable gate arrays (FPGA), and extensions to recent PCs, for example, the Multi-media Extension (MMX) [FKO+04, HIG02] and the pixel/vertex shading instructions for the Graphics Processing Units (GPUs) [GY05, MPL05].

3.5/ Related works on stereo-matching problems

3.5.1/ Current progress in the real-time stereo computation

From the beginning of 1990s, real-time dense disparity map stereo has become a reality, making the use of stereo processing feasible for a variety of applications. Until very recently, all near real-time implementations made use of special purpose hardware, like digital signal processors (DSP) or field programmable gate arrays (FPGA). However, with ever increasing clock speeds and the integrating of single instruction multiple data (SIMD) coprocessors, such as Intel MMX, the NOC and the Graphic Processing Unit (GPU) into general-purpose computers, near real-time stereo processing becomes a reality for common personal computers. This section presents the progression of fast stereo implementation over the last two decades.

In 1993, Faugeras *et al* reported on a stereo system developed at INRIA and implemented it for both DSP and FPGA hardware [FHZF93]. They implemented normalized correlation efficiently using the block matching method with left-right consistency checking. They used a right-angle trinocular stereo configuration, computing two estimated disparity maps and then merging them to enforce joint epipolar constraints. The DSP implementation exploited the MD96 board [Mat93], that consists in four Motorola 96002 DSPs. The other implementation on FPGA was designed for the PeRle-1 board; which was developed at DEC-PRL and was composed of 23 Xilinx logic cell arrays (LCA). The algorithms were also implemented in C for a Sparc 2 workstation. The results showed that the FPGA implementation outperformed the DSP implementation by a factor of 34 and the Sparc 2 implementation by a factor of 210, processing 256×256 pixel images at approximately 3.6 fps.

In the same year, Nishihara presented a stereo system based on the PRISM-3 board developed by Teleos Research [Nis93]. This system used Datacube digitizer hardware, custom convolver hardware, and the PRISM-3 correlator board, which makes extensive use of FPGAs. For robustness and efficiency, this system employed area correlation of the sign bits after applying a Laplacian of Gaussian filter to the images. Two years later, Konolige also reported on the performance of a PC implementation of these algorithms by Nishihara in 1995 [Kon97]. His system was capable of 0.5 fps with 320×320 pixel images.

Also in 1993, one system came close to achieving frame-rate. Kanade *et al*'s algorithms, the multi-baseline stereo method, was used by WEbb implemented on the CMU warp machine [CW91, KON92, Web93]. Three images were used for this system. For efficiency, the sum of absolute differences (SAD) was used. They used 64 iWarp processors and could achieve 15 fps with 256×240 pixel images.

In 1995, Matthies *et al* presented a real-time stereo-matching system using a Datacube MV-200 image processing board and a 68040 CPU board [MKLT95]. Including the left-right consistency checking, this performed SSD stereo-matching on a Laplacian image pyramid to determine disparities. They could achieve 1.7 fps with 256×240 pixel images. A complete discussion of their algorithm development and details on an earlier version of the system can be found in [Mat92]. Also in 1995, Kimura *et al* reported on a video-rate stereo machine developed at CMU [KKK+95]. This was the first published stereo system capable of more than 24 fps with 256×240 pixel images. Like the iWarp implementation at CMU, this system also exploited multi-baseline stereo to improve depth estimates. The prototype system was equipped with six cameras. Also, like the iWarp implementation, the SAD was used for efficiency. These algorithms were implemented on custom hardware and an array of eight C40 DSPs. The CMU video-rate stereo machine has been used for a variety of applications, including virtual reality [KYO+96].

Two years later, Woodfill and Von Herzen [WV97] implemented the famous census matching [ZW94] for stereo on the custom PARTS engine developed at the Interval Research Corporation. The PARTS engine is made up of 16 Xilinx 4025 FPGAs and fits on a standard PCI card. It is capable of processing 320×240 pixel images at 42 fps. Corke and Dunn [CD97] also implemented census matching on their Configurable Logic Processors (CLP), VMEbus circuit boards made up of several FPGAs each. Their system is capable of processing 256×256 pixel images at 30 fps.

From 1998 to 2005, new approaches were still reported in the stereo-matching field [KSY+99, BK99, DGHW98], but people were turning to some platforms more usually equipped on PC.

In 2005, Heiko Hirschmuller [Hir05] presented an accurate semi-global stereo-matching method concentrating on the object boundaries. This method can perform pixel-wise matching based on mutual information and the approximation of a global smoothness constraint. It was capable of 1 fps in performing typical images. In the same year, Veksler [Vek05] implemented the dynamic programming (DP) into stereo-matching problem. By applying the DP to a tree structure, where the nodes of the tree are all pixels, this method could handle a typical image in less than 1 second. Also by the DP, [WLG+06] which integrated an adaptive aggregation step in the DP stereo frame-work, the matching cost is aggregated only in the vertical direction, and this implementation has come to over 50 million disparity evaluations per second.

Tombari *et al* [dGGV08] proposed a new cost aggregation strategy that shapes the carriable support at each correspondence based on information derived from image color segmentation. This strategy could be capable of 5 fps in processing typical size (320×240) images.

Owning to the complexity of global methods, most of them are not suitable for real-time applications. To the best of our knowledge, there are only two global methods that can be found running in real-time. The first is proposed by Forstmann and Kanou [FKO+04]. It is a real-time stereo system based on dynamic programming and it consists in a specific coarse-to-fine approach in combination with MMX implementation which uses compiler

optimization strategies to achieve real-time stereo-matching performance. Another algorithm is proposed in [GY07] based on dynamic programming. In this algorithm, the iterative best path tracing process used by traditional dynamic programming is replaced by a local minimum search process, making the algorithm suitable for parallel execution.

Nevertheless, local Winner-Takes-All (WTA) optimization is still the most used one in almost all real-time stereo-matching algorithms. Faugeras *et al* [FHZF93] has proposed a sliding window technique for rectangular aggregation windows of fixed size. This method has linear complexity with respect to the numbers of pixels and computed disparities, which gives rise to real-time implementations such as that of Point Grey Research [1]. A detailed review about real-time stereo-matching algorithms based on local methods can be consulted in [WGY06].

All these methods are executed on CPU or some special designed hardware. And indeed, they have performed quite well for fast stereo-matching. However, with the raise of the integrated hardware, such as the GPU, more and more people are attracted to this new implementation field, and the parallel computing becomes a popular subject for researchers.

3.5.2/ PARALLEL COMPUTING BASED ON GPU IN STEREO-MATCHING

Most of the latest near real-time stereo-matching methods are based on parallel computing, especially on GPU. The GPU has been employed in stereo-matching problem for almost ten years. Many methods have been introduced for last decades [Mei11, SWP+12, WLG+06, GGK09]. Several implementations are available in the literature.

Researches on GPU parallel computing in stereovision can be roughly classified into two classes, the first one is the proposition that takes the running time more important than matching accuracy or tends to look for a trade between the computation time and matching quality. The CostFilter[RHB+11], PlaneFitBP[YEA08], ADCensus[Mei11] belong to this class. While the second class consists of the methods which concentrate on accurate stereo-matching, and the use of GPU is only a way to accelerate computation [HBG10, LS10, WTFJ10].

In 2008, Yang *et al* [YEA08] presented a compromising approach for stereo depth estimates. It can replace estimates in textureless regions with estimates on planes at near real-time rates. Their approach segments the image via a novel real-time color segmentation algorithm, and subsequently fits planes to textureless segments and refines them using consistency constraints. Moreover, the authors have optionally employed loopy belief propagation to correct local matching errors. This method can handle 18 frames per second using an NVIDIA Geforce 8800 GTX graphics card, and it remains one of the top three best implementations in the Middlebury database [HS06].

Besides PlaneFitBP, Rhemann *et al* [RHB+11] proposed in 2011 a genetic framework containing three steps: constructing a cost volume, fast cost volume filtering and winner-takes-all label selection. By this simple framework, they have obtained the disparity map in near real-time, whose quality exceeds those of all other fast (local) approaches on the Middlebury stereo benchmark. In the same year, Mei *et al* [Mei11] presented a GPU-based stereo-matching system including an AD-census measure. It uses the cost aggre-

[1] http://www.ai.sri.com/~konolige/svs/svm.htm

gation in dynamic cross-based regions. [ZWC+13a] reported a GPU-based framework with SAD-AL stereo measure, adaptive window cost aggregation strategy and a simple refinement step. Both these two approaches can handle the typical images in less than 0.1 seconds, more specially, the ”Tsukuba” image in less than 0.02s. And the ADCensus, achieving excellent trade between the computation time and the accuracy, is still one of the best implementations in Middlebury benchmark.

Some researchers have introduced the accurate stereovision approaches from CPU to GPU, and these methods can handle the matching work of well quality with some acceleration in comparing with sequential computation on CPU. In 2010, three approaches were proposed. The first one was reported by Asmaa *et al* [HBG10]. Their method exploits the adaptive support weight windows for generating an explicit per-segmentation of the reference image in a fast way. By using a modified segmentation-based sliding window technique which makes run time independent of the window size, they achieve 10 fps for the four benchmarks in Middlebury database. Yu *et al* [WTFJ10] proposed a system based on GPU with the Pareto-efficiency front-line in the accuracy and speed trade-off space but aiming at high matching quality. They have designed an exponential step size message propagation (ESMP) which incorporates the smoothness term commonly used in belief propagation for global stereo-matching methods. This approach can handle a stereo image pair of size 384×512 in 0.096s and be able to demonstrate a speed-up factor of 2.7 to 8.5 in common stereovision applications.

3.6/ CONCLUSION

In this chapter, the basics of stereovision have been described. Details about the stereo-matching problems consisting in finding out pairs of homologous pixels in the left and right input images have been presented. We have introduced the important cases where stereo-matching schemes fail, such as the textureless region and the repetitive regions in the images. We have explained the useful assumptions about order and half-occlusion. Finally, the current progress of parallel computation in stereo-matching problem has been introduced. As in this document the two matching methods used are both local dense stereo-matching methods, so we describe the details of the local dense matching methods. This class of methods analyzes the neighbors of the center pixel and the neighbors of its candidates to identify pairs of homologous pixels. The local methods are less time consuming, in comparing with global stereo-matching methods which evaluate matching globally with all the pixels in the images. One important method in local dense stereo-matching is winner-takes-all method, which can be easily implemented in parallel architectures such as multi-processors or GPUs. We will investigate different GPU implementations of this general method, by combination of diverse techniques generally simulated in CPU platforms.

4

BACKGROUND ON SELF-ORGANIZING MAP AND STRUCTURED MESHING PROBLEMS

4.1/ INTRODUCTION

In this chapter, we will focus on the background of the self-organizing map used for structured meshing problems. A structured mesh is generally a grid of vertices having a fixed local neighborhood. Structured meshes correspond to the three possible tessellations of the plane with identical regular polygons: square, triangular, and hexagonal. Mesh generation is a large area of research and 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 also a meshing technique useful for surface reconstruction. In that case, the advantage can reside in the grid data structure with fixed dimensions, and constant neighborhood that allow fast access computation and compact representation. The self-organizing map (SOM) algorithm is a neural network approach dealing with visual patterns moving and adapting to brute distributed data in space. It is a usual tool for structured meshing generation in the plane or 3D space. Its main property is to allow adaptation by density and topology preservation of a planar graph to an underlying data distribution. These properties are the requisites that allow to address structured meshing and even solve traveling salesman problem by using such a simple and massive parallel heuristic.

We propose in this document a GPU parallel implementation for SOM in 2D space and its applications to two domains: meshing for stereovision in image processing, and meshing for terrestrial transportation problems with the example of TSP. Our intention is to show that the SOM algorithm can be used as an heuristic, or basic tool operator, useful in both domains. In this chapter, we make a survey of SOM for structured meshing applications and surface reconstruction in relation to stereovision. Secondly, we present the concept of adaptive meshing for transportation applications, using SOM as a central tool, that were already developed for Euclidean routing problems. Finally, we provide related works particularly on parallel computing on the Euclidean TSP. For each case of application, we look specifically at the GPU models already used in the literature.

4.2/ SELF-ORGANIZING MAP FOR GPU STRUCTURED MESHING

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, relative to the underlying resource. Numerous industrial applications need an optimization of calculation points in order to minimize the computation time or to maximize the resulting precision (fluid mechanic applications, interference calculation for frequency assignment...). Most of the time, the initial data subject to the computation is situated like in geographic coordinates and altitudes for meteorological measurements, and it can be regularly distributed or not. While mesh generation is a large area of research [Car97, Geo91, dGGV08, PSB+07], few approaches exist for structured mesh generation according to a density distribution, in order to represent data by density and topology preservation. Clearly, it is the main property of the self-organizing map [Koh01, Koh82] algorithm that it can generate such structured meshes. For example, [Bon89, SST91] deal with this subject but the implementations were sequential and not competitive. Numerous approaches like [Sch00, ZB06] have proposed generation methods for structured meshes but the methods are often too memory consuming and can not handle large size problems or are not sufficiently fast for real-time execution. It looks like that GPU parallel computation is still an open field for such problems and applications.

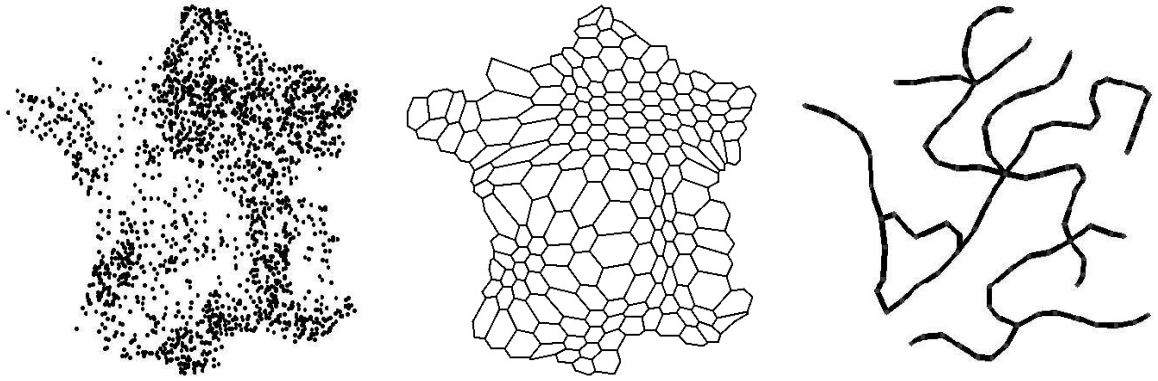
Considering SOM applications on structured meshing problems, most of applications do not refer to parallel execution or real-time requirement. In [PCA98] the authors introduce an approach to the local stereo matching problem using edge segments as features with several attributes. They use a learning strategy based on the self-organizing feature-mapping method to get the best cluster centers. The authors of [BEP05] propose an original neural network architecture inspired from Kohonen's self-organizing maps, based on adaptive learning process applied to a generalized mesh structure that would lead to a coherent topological definition of the surface, represented by a points cloud, given as input. However, they still need to find the fine tuning concerning the local adaptation and subdivision process. [LWC03, Nec10, dRAN07] integrate the self-organizing map with meshes and surface reconstruction but also they never refer to the cellular space decomposition for further studies of parallel implementations. [CB08] proposes a simple combination of self-organizing map and cellular automata in meshing processing, but the paper does not explore deeper neither in the properties of this combination nor the possibility of execution on parallel architectures. In [YIL08], the authors present a SOM based algorithm for implicit surface reconstruction. But no parallel GPU implementation is provided by the authors. The authors in [VGB09] design a new stereo-matching model based on SOM, aimed at solving the correspondence problem within a two-frame area matching approach and producing dense disparity maps. Salient aspects of the solution were the local processing of the stereo images, the use of a limited set of directly available features and the applicability without image segmentation.

From our knowledge, we did not find GPU parallel implementations for structured meshing applications based on noisy distribution maps, except simple Delaunay triangulation based on a point set. We did not find GPU parallel implementations for the SOM algorithm when applied in 2D space. Some methods for computing SOM on GPU have been proposed [MSH+12, YKN+12]. All these methods accelerate SOM process by parallelizing the inner steps in each basic iteration, and mainly focus on two aspects: finding winner neuron and moving the winner neuron as well as its neighbors in parallel. In our model, we use parallel processing units 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.

4.3/ ADAPTIVE MESHING FOR TRANSPORTATION PROBLEMS

The origin of adaptive meshing which uses planar honeycomb structures as a tool to dimension radio-cellular network according to mobile traffic is from [CLK00, CKLC05, CK06]. Then, the concept was generalized for transportation routing problems by using the SOM algorithm as a common tool. This section presents the adaptive meshing concept and summarizes several applications already developed for distributed terrestrial optimization problems [CK07a, CK07b, CKH07, HCK05].

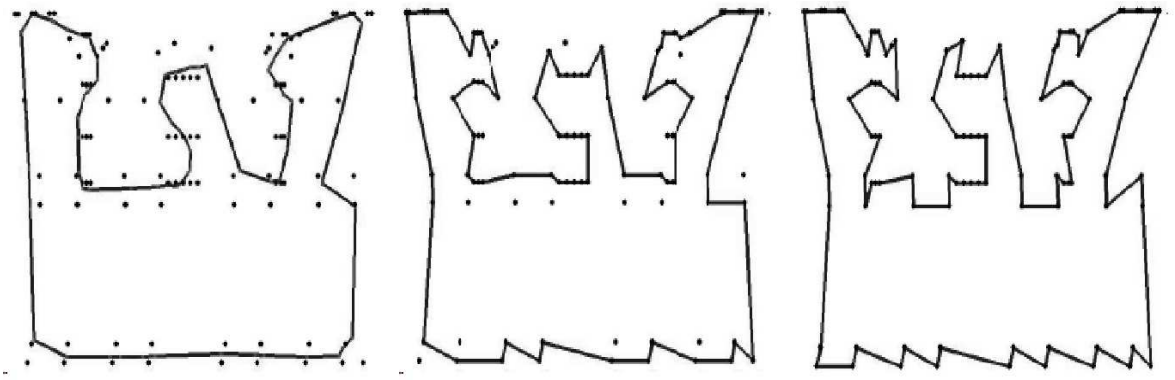


(a) (b) (c)

Figure 4.1: Meshing of a territory. (a) Sampling of the demand (1000 dots) on a territory. (b) Adapted honeycomb mesh representing a radio-cellular network. (c) Adapted graph of interconnected routes representing a transportation network.

In [CKLC05, CK06] the authors have taken the visual specificity of SOM and the adaptive meshing concept into radio-cellular networks. The goal is to adjust an intermediate structure (the network) to an underlying distribution of demands, shown in Fig. 4.1(a), subject to topology constraints. This is illustrated for cellular network dimensioning in Fig. 4.1(b), with a honeycomb mesh representing cell transceiver base stations covering a territory. The terrestrial transportation case is illustrated in Fig. 4.1(c), where interconnected lines stand for interconnected routes.

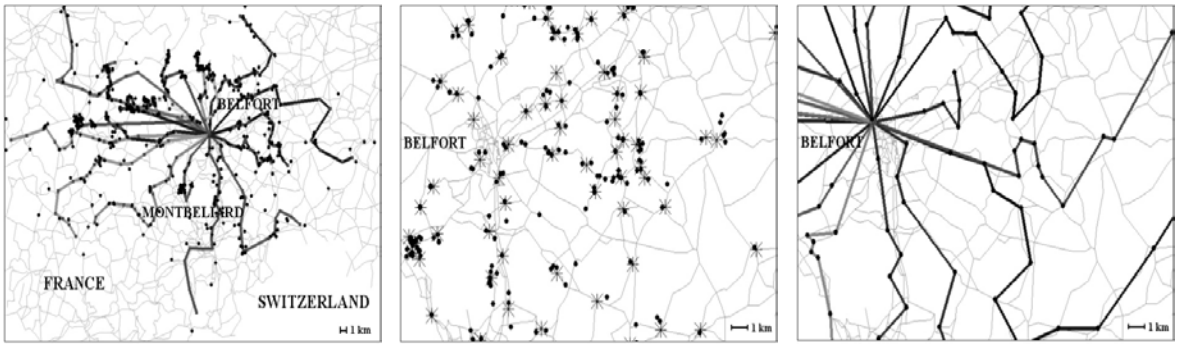
As another application of adaptive meshing, the Median-Cycle Problem (MCP) has been investigated by [LLRG04, LLRG05, RBL04]. It consists of finding a ring passing through a subset of cities, minimizing objective length, subject to a bound on distance to clusters. Application of SOM to the problem consists of using a ring structure, the mesh, that adapts to the point set. The memetic SOM approach [HCK05] was developed to solve the problem. It consists of a genetic algorithm including SOM as a specific operator. Solution for the MCP is illustrated in Fig. 4.2, on the lin105 instance of TSPLIB [Rei91]. In Fig. 4.2(a), is shown a Euclidean, or continuous, version of MCP, where the ring has cluster centers free to position anywhere in the plane. In Fig. 4.2(b), cluster centers are forced to be located on point locations, as in the case for the classical and discrete MCP. In Fig. 4.2(c), with a same number of centers and cities, MCP becomes a TSP.



(a) (b) (c)

Figure 4.2: Median cycle problems (MCP) using the lin105 TSPLIB problem. (a) Euclidean MCP. (b) Classical MCP. (c) Euclidean TSP.

In [CK07a, CK07b], the adaptive meshing approach is illustrated on a real life case of combined clustering and vehicle routing. The goal is to locate bus-stops on roads and define the regional routes of buses to transport the 780 employees of a great enterprise. In Fig. 4.3, are shown visual patterns of solutions. A transport mesh obtained is shown in Fig. 4.3(a), juxtaposed over demands and underlying roads. It illustrates the visual shape of a typical solution, where bus stops, reflect demand distribution and constitute routes. In Fig. 4.3(b) and (c), a zoom is performed on the right side of the area to illustrate the two main steps for solving the problem sequentially. Fig. 4.3(b) presents bus stops, symbolized by crosses, obtained for k-median problem. Fig. 4.3(c) presents the vehicle routing problem solution obtained subsequently, with routes exactly passing by the crosses.



(a) (b) (c)

Figure 4.3: Clustering and routing for the transportation of 780 customers (dots) of a great enterprise. (a) Unified clustering and routing problem solution. (b) Clustering k-median problem first (crosses are cluster centers). (c) Capacitated vehicle routing problem second (routes pass among cluster centers).

In the literature, many applications of neural networks have addressed the traveling salesman problem. For more information on this subject, we refer the reader to the extensive survey of [CB03]. In [CK09], the authors evaluated the memetic SOM performance on

the Euclidean TSP. They compared it against the Co-Adaptive Net of [CB03], which was considered at the date of writing as the best performing neural network application to the TSP. Experiments were conducted on large size TSPLIB instances from 1000 to up 85900 cities. Mainly, numerical results tended to show that memetic SOM competes with the Co-Adaptive Net, with respect to solution quality and/or computation time.

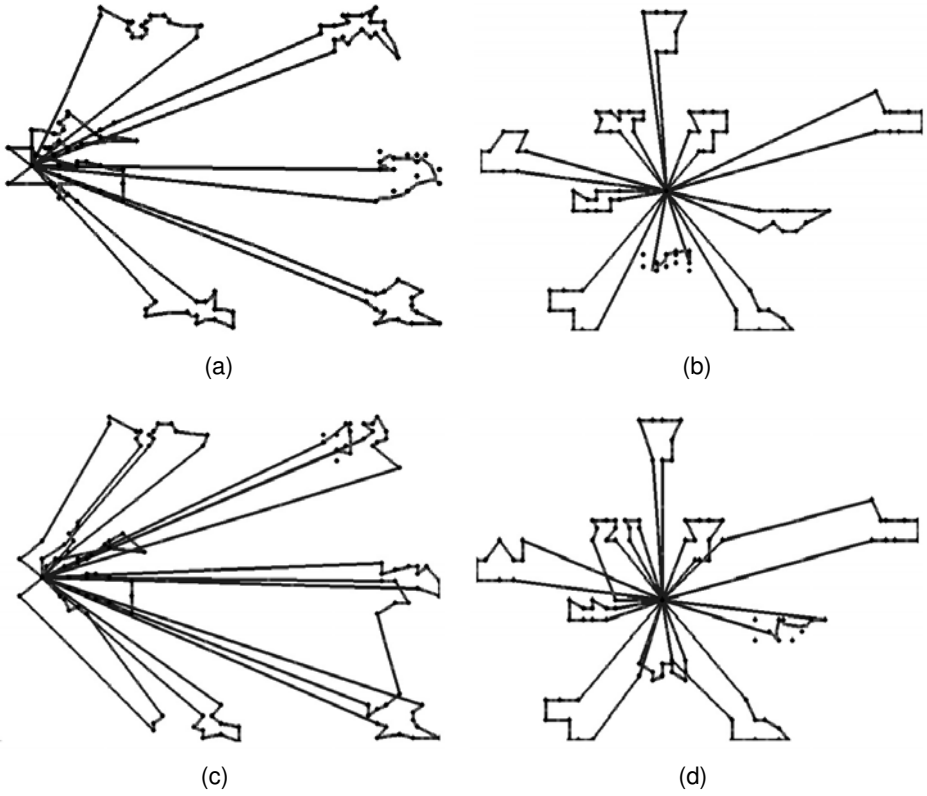
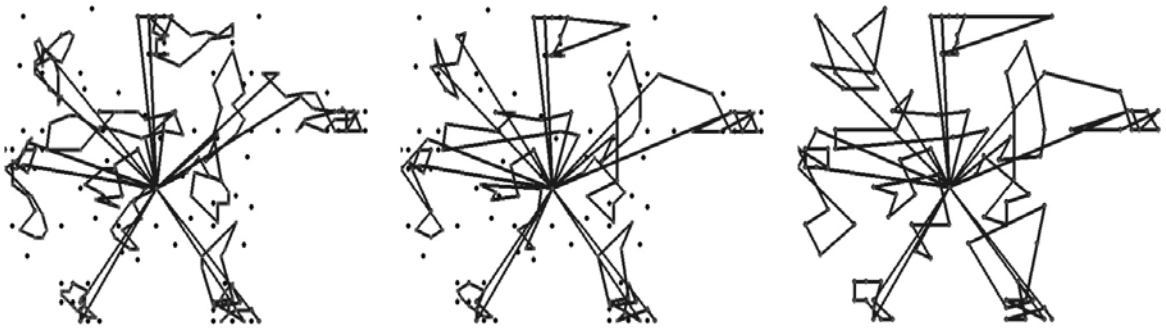


Figure 4.4: VRP using c11 (a) and c12 (b) instances. VRP with time duration constraint using c13 (c) and c14 (d) CMT instances.

The memetic SOM approach was applied to vehicle routing problem (VRP) in [CK07b]. Example of local improvement moves performed during the optimization phase are visualized in Fig. 4.4 on the clustered instances c11-14, from the publicly available Christofides, Mingozzi, and Toth (CMT) test problems [Chr76].

In [CKH07], a memetic SOM application to address a clustering version of the vehicle routing problem with time windows (VRPTW) is proposed. The approach was tested on Solomon's standard test problems [Sol87]. Visual patterns in Fig. 4.5(a-c) illustrate solution on the rc201 test case.

Subsequently, [CHKK12] illustrates how the concept of adaptive meshing, provided by the original SOM, is naturally suited for application into a dynamic setting, specifically to address the dynamic VRP. The experiments show that the memetic SOM method performs better with respect to solution quality than an ant colony algorithm MACS-VRPTW, a genetic algorithm, and a multi-agent oriented approach, with a computation time used roughly 100 times lesser.



(a) (b) (c)

Figure 4.5: Clustering Vehicle Routing Problem with Time Windows, using the rc201 Solomon test case. (a) Obtained solution. (b) Same solution after removing empty clusters. (c) A classical VRPTW solution obtained after projecting cluster centers to their (single) assigned requests.

4.4/ Parallel computing for Euclidean traveling salesman problem

4.4.1/ Solving TSPs on general parallel platforms

The Euclidean traveling salesman problem (TSP) is a classical and widely studied combinatorial optimization problem. The problem is NP-complete [Pap77]. It consists of finding a shortest tour passing to a set of N cities located in the plane, such that each city appears in the tour only once. For years, a lot of research has been done in the TSP field, a detailed literature on heuristics and metaheuristics methods can be found in [JM02, JM97]. It appears that the best performing approaches for TSP are specific and very sophisticated implementations of simple local search methods such as 2-Opt, 3-Opt, and Lin-Kernighan (LK) and that metaheuristics like tabu search of genetic algorithms are time consuming methods. Compared to other metaheuristic approaches, memetic algorithms that combined local search into population based genetic algorithm look to be superior. However, very few metaheuristic approaches appear to be competitive with the Helsgaun [Hel00] implementation of the LK heuristic, on both computation time and solution quality. As far as we know, the memetic algorithm of Nguyen [NYYY07], that already includes LK as a main embedded local search, looks very competitive, even for large size problems. The simpler Iterated LK heuristic appears to be the better choice if good solutions are required in short time. Then, a difficult question to answer is how to adapt such complex powerful heuristics for the TSP to the multi-processor parallel platforms.

A lot of applications have studied parallelism of work stations or clusters to accelerate algorithms for TSP. They can be often based on genetic algorithm or data and subdivision partition. For example, the studies and applications [Kar77, Kar77, PMM88, KL89, JK90, Ces96, SOS97] more or less investigate partition and subdivision of data. They subdivide the set of input cities into smaller sets, and compute an optimal sub-tour for each subset. These sub-tours are then combined to obtain the tour for the entire problem. Different studies of coarse-grained parallel genetic algorithmic can be found in [WmS+05, NYYY07, Bor06]. Such methods can also be combined with Lin-Kernighan local search heuristic as in [BHP01, Hel00]. These methods are generally executed on a cluster of workstations

with high granularity.

4.4.2/ SOLVING TSPS ON GPU PARALLEL PLATFORM

The most frequent GPU applications to TSP use genetic algorithm (GA) or ant colony optimization (ACO) [Dor92] that are both population based metaheuristics. Other GPU implantations consider parallel computation for neighborhood operators in simple local search.

Population based methods such as AG and ACO are good candidate for parallel execution when considered at a high granularity level, where parallelism takes place at level of a single solution. Then, parallelisms resides in the multiple evaluations and operations performed independently on each individual (solution) of the population. Parallelism resides in data duplication and not in input data partition. Some examples of GPU implementations based on GA are proposed by [YCP05, DMMC09, CDJN11, PJ09]. The study of ACO's parallel algorithms has been figured out by [Stü98, TRFR98]. In years, several ACO GPU implementations have been proposed as [You09, WDZ09, CGN+13, DDGK13]. Such GPU implementations can reach an acceleration factor as big as 20 for small size TSPs in comparison to the sequential version on CPU. However, the methods have the common weakness in memory employment. These population based methods need significant memory space to store the solutions, so when the instance size increases, the memory space needed will go beyond the hardware's capability. In a device with limited memory, the instance size should diminish as the number of processing units augments. This limits the performance of this model for large size instances.

Some GPU implantations that compute neighborhoods for local search or tabu search methods are also proposed as in [JJL08, FT11, Luo11] for solving TSPs. Generally, to each thread is associated the computation of some neighbors in neighborhood. Because of the large size of neighborhoods that grows at least as the square of input size (for 2-Opt specially), it is a very difficult task to assign the processors to the neighbors. They consume a lot computing resources and larger size problems with more than five thousand cities stay always the endpoint for these methods. Moreover, from our knowledge, we did not find GPU implementation of SOM to the Euclidean TSP in the literature, and no GPU implementations for large size TSP problems with up to thirty thousand cities, as we present in this document.

4.5/ CONCLUSION

We have presented some background and literature survey on GPU implementations of SOM algorithm, and application to structured meshing in stereovision and to routing problem as TSP. We did not find GPU implementation of SOM in 2D and 3D space, as a distributed and decentralized algorithm as we propose. It looks that current GPU parallel models for mesh generation mainly focus on triangulation methods. Also, current GPU models for solving TSP have difficulties in handling large size instances, and their shortcomings resides in important memory use.

One important point of the new parallel model for SOM presented in this document, being able to address the two problems of meshing and routing, is that it proceeds by a cellular decomposition of the data, i.e. the disparity map or the set of cities in the plane, such

that each processing unit represent a constant and small part of the data. Hence, the approach should be more and more competitive according to the increase of the physical cores, and be able to deal with very large size problems at the same time.

II

CELLULAR GPU MODEL ON EUCLIDEAN OPTIMIZATION PROBLEMS

5

PARALLEL CELLULAR MODEL FOR EUCLIDEAN OPTIMIZATION PROBLEMS

5.1/ INTRODUCTION

In this chapter, we will center our attention to present the proposed cellular GPU parallel model for massive parallel computation on some Euclidean optimization problems. Firstly, we detail the partitioning of the problem input data, called distribution map, between a set of cells organized in a two-dimensional cellular matrix and that define the computational level of granularity. To each cell, is assigned a single processing unit. While data decomposition is not a new way of dealing with parallelism in general, here, we specifically assume a linear association from input data to computation cores as the problem size increases, when dealing with Euclidean optimization problems. This is the main assumption that makes us speak about massive parallelism, and allows to address large size problems. Each cell is responsible for a constant part of the data. It is also assumed that distributed computation between processors has no central control, unlike most of current metaheurisctics on GPU that use GPU calls from a sequential master algorithm. We present and discuss the main characteristics of the approach, its advantages and shortcomings, with regards to standard GPU approaches in combinatorial optimization.

Secondly, we give the principle of the model application to two types of problems. The first type are local dense stereo-matching problems. The solution partition is standard in that, each pixel is already a basic processing unit. Parallel local dense winner-takes-all methods will be applied for that problems. Second type of problems are Euclidean clustering problems, called meshing problems here, that include the standard Euclidean traveling salesman problem. They are addressed by using the self-organizing map procedure as a simple and distributed heuristic for that problems. The continuity between the applications is that meshing is applied to a disparity map obtained from stereo-matching, and that TSP solution is also a mesh in the plane. Both approaches and applications share in common distributed and massive parallelism at cellular level.

This chapter contains three main sections. The first section presents the principle of data and treatment decomposition within a matrix of cells and threads, and the main characteristics of the general parallel computation model. The second section discusses its application to stereo-matching, even if the cellular decomposition is straightforward since one pixel stands for one cell. The third section details the cellular parallel computation model for the self-organizing map procedure that we specifically propose, and its application to meshing generation according to a disparity map, and traveling salesman problem.

5.2/ DATA AND TREATMENT DECOMPOSITION BY CELLULAR MATRIX

Here, we expose the main characteristics of the parallel cellular model proposed to address some Euclidean optimization problems. In such problems, the input data of the problem are numerical entities distributed onto a surface or plane. We assume that the input data is given by a distribution map representing some population of values over the plane. Such distribution map, also called density map, can either be a matrix of pixels for image processing or a set of points representing requests or customers located in the plane for transportation applications.

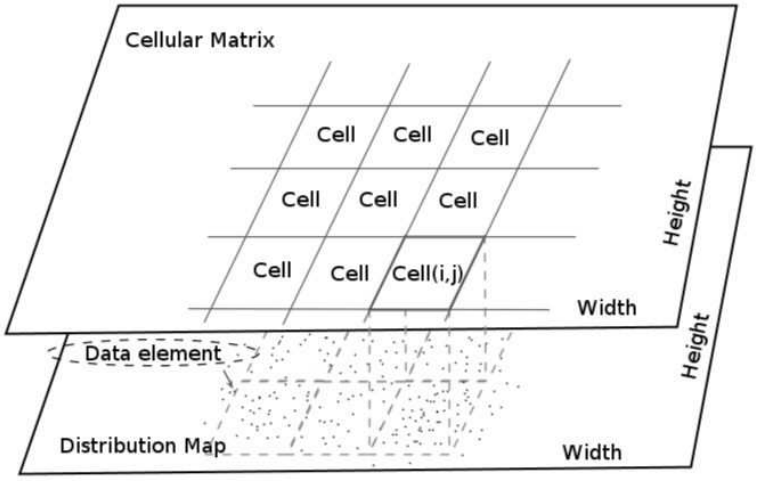


Figure 5.1: Partitioning of the input distribution map.

Then, based on such distribution map, one important point of the parallel computation model presented in this document, is that it proceeds by a cellular decomposition of the data, where each cell of the decomposition is necessarily assigned to a single processing unit, or thread for GPU platform, and represents a constant part of the input data according to the problem size.

Fig. 5.1 illustrates the partitioning. In this figure, the input data distribution map lying at the bottom are partitioned into a matrix of cells, represented in the figure as map of cells juxtaposed to the former. A data element from the input distribution map can be a pixel or more simply, a point in the plane. To each cell corresponds a single thread that will compute locally based on its cell contents as well as on the contents of its local neighborhood cells. It could follow from such neighborhood interactions concurrent accesses to shared data.

An important assumption of the model is that a given cell represents a constant and small part of the input data. Hence, according to the increase of physical parallel processing units in the future, the approach should be more and more competitive, while at the same time being able to deal with larger size problems. This property holds true because of the linear memory needed corresponding to the distribution map size. We will illustrate that point in experiments by systematically analyzing the influence of the input data size on performances, and dealing with large size instances.

The parallel cellular model offers some advantages in comparing with some current paral-

lel optimization approaches on GPU. For example, when looking at the GPU applications to the standard Euclidean traveling salesman problem, the main optimization approaches executed on GPU are population based metaheuristics, like the genetic algorithm or ant colony, or neighborhood search methods. Since population based methods are based on data duplication, the more is the population size, the less is the instance size contained in a standard GPU with limited memory size. For the neighborhood search, that is based on parallel computation of the solution neighbors, it is worth noting that the fast increase of neighborhood size limits the size of the problem being addressed. For example, a simple 2-opt neighborhood operator already generates $O(N^2)$ neighbors. According to recent literature on GPU applications, it looks that the larger size problems stay as the endpoint for these models. We do not see GPU application to the traveling salesman problem using more than about 5,000 cities. Here, we will address an instance with up to about 30,000 cities. However, the classical models are generic methods, complementary to the proposed cellular method. It should be possible to use cellular parallel computation into a population based framework, or combine classical neighborhood operators with it. Also, one can execute local search based on a linear decomposition of the data. Another difference, is that current approaches are mixed sequential/parallel methods, whereas our method addresses distributed computation between cells with no central control. Cellular genetic algorithms also implement distributed control, but the model presents hight granularity with input data duplication in population of solution. Also, classical methods necessitate a construction method, to first build an initial solution that will be improved by the metaheuristic. In our case, the method generates a solution from scratch.

The parallel cellular model presents some shortcomings. It is first restricted to Euclidean problems and specific classes of problems. Here, we address local dense stereo-matching problems, from the one hand, and meshing problems, that are clustering problems in the plane with topological relationships between cluster centers, on the other hand. There exist interactions and concurrences betweens threads in accessing memory space. In stereo-matching problems, the matching cost aggregation needs added density values from its neighbors in its support region. While in the meshing problems, the grid nodes of the meshing share some common parts, and are concurrently accessed by neighborhood threads. All these interactions and concurrences in memory access need atomic operations to coherently update the memory, but in a sequential order. The behavior of the model can depend on the data distribution. As the data distribution may present great variations of spatial density, it should be the case that some cells would have a more important computation work than others. This introduces divergence between thread execution and makes computation serial. It is worth noting that uniform distribution corresponds to the most balanced cellular decomposition and hence to equilibrated multi-processor load. This is what we call ”massive parallelism”, the theoretical and ideal possibility to execute $O(N)$ simultaneous parallel operations, with N the problem size, in average for uniform or bounded distributions. We now present our four applications, that are also referred in [ZCW+13, ZWC+13a, ZWC+13b, WZC13], of that cellular parallel model.

5.3/ APPLICATION TO STEREO-MATCHING

In the previous section, we have introduced the cellular parallel GPU model, and presented the details of the partitioning of the input distribution maps. We now present the

stereo-matching methods that we have implemented on GPU. We present the parallel model, the two applications addressed, and recall the winner-takes-all method.

5.3.1/ LOCAL DENSE STEREO-MATCHING PARALLEL MODEL

In local dense stereovision, the stereo-matching is based on local pixel intensity values of the input image pairs. The methods for these problems assume that the density and intensity configurations are similar in the neighborhood of homologous pixels. So, a matching cost is defined between the window around the left pixel to be matched and the window around the candidate pixels in the right image. The window is also called support region. Between local dense matching methods, the differences are mostly in the matching cost computation, and in aggregation of the matching cost in support region for each pixel. Generally, based on the support region, methods can be classified into two branches: the fixed window methods, whose support region has a fixed shape and dimension, and the adaptive window methods, in which the support region pixel can represent the texture segment in its neighborhood. We will investigate the two types of methods.

The cellular parallel computation model used for stereo-matching problems is simply the pixel decomposition level of the image. The model is straightforward in GPU platforms for image processing. The input distribution maps are mainly the left image and the right image. For every input image, the partition we used for the cellular matrix is then the pixel partition itself. Each pixel corresponds to a single cell of the parallel model, and to a single thread. Each thread is in charge of one pixel, and the thread will handle all the necessary operations at the pixel location, such as computing window aggregated costs between neighborhood pixels.

In this document, we propose stereo-matching GPU implantations for two methods and their variants: a CFA stereo-matching method with fixed window size, and a near real-time stereo-matching method based on adaptive-window cost aggregation. Both methods use the winner-takes-all selection method of homologous pixels.

5.3.1.1/ CFA STEREO-MATCHING APPLICATION

This first approach is a naive and straightforward implementation of the CFA stereo-matching method originally proposed by Halawana [Hal10] based on winner-takes-all method, applied on two color components only. The application proposed in this document extends the experiments and evaluations originally proposed in [Hal10]. We systematically study influence of image size, and the trade-off between quality and computation time, on both CPU and GPU and extend comparisons with other similar methods.

In the acquisition of the color images, two types of cameras can be used: the cameras that are equipped with three sensors associated with beam splitters and color filters providing the so-called full color images whose pixels are characterized in Red, Green and Blue levels, and the cameras which are equipped only with a single-sensor. In the second case, the single-sensor can actually deliver a color filter array (CFA) image, of which every pixel is characterized by a single color component, which can be one of the three color components: Red (R), Green (G) and Blue (B). So, the missing color components have to be estimated at each pixel. This process to estimate the missing color components is usually referred as CFA demosaicing and generates a demosaicing color image,

where every pixel is presented by an estimated color point. The CFA stereo-matching problem refers to the stereo-matching applied to CFA-images as input. The originality of the Halawana method consists in only estimating the second color component of a pixel, then avoiding the use of the third one.

5.3.1.2/ REAL-TIME STEREO-MATCHING APPLICATION

The real-time stereovision is an important branch in stereo-matching problems, since the constraint on computation time is very important for applications that run at video rate. So, the running time of stereo-matching methods used should be less than 34 ms which is generally the image acquisition time in filming the videos. Despite a substantial GPU acceleration over CPU, the previous parallel method used for CFA-images was not able to reach real-time computation time. What we have to do is to find out one efficient matching strategy to pick out the right homologous pixel in a set of candidates, which is the most time-consuming part in stereo-matching processing. To improve quality results, and also reduce computation time, we investigate memory organization, adaptive window strategy and a last refinement step to eliminate out-layers.

5.3.2/ STEREO-MATCHING WINNER-TAKES-ALL METHOD

While stereo-matching problems presented in their general form are NP-hard problems [KU03, Vek99], this does not prevent from using very simple search strategies, as it is the Winner-Takes-All (WTA) principle. In the stereo-matching process, to each pixel of the left image is associated a thread. Then, the thread has simply to find out the best pixel in a set of candidate pixels in the right image, as the homologous pixel for the pixel in the left image. The data can be illustrated by Fig. 5.2 and the thread execution process is presented in Algorithm 1. In the figure, given a pixel x_l in the left image, a set of candidate pixels in the epipolar line in the right image are chosen. Then, for each of these candidates, the aggregated matching cost is computed between the candidate and the referred pixel x_l, taking care of the window region around pixels. With the comparison among the computed matching costs for all the candidates, the homologous pixel in the right image is derived based on the WTA principle. The shift for which the matching cost is the lowest is selected. Thus, the estimated disparity $\hat{d}_l^w(x_l, y)$ at the pixel x_l corresponds to the shift *s* of the right pixel, at which the matching cost is the minimum. It can be expressed as Equation 5.1, here with the SSD matching cost.

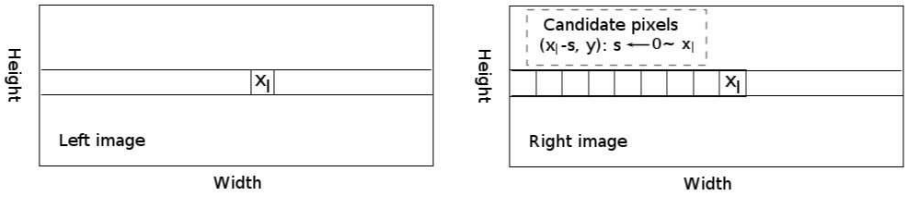


Figure 5.2: Given one pixel in the left image, picking out the best candidate pixel in the right image.

$$\hat{d}_l(x_l, y) = arg\min_s(SSD_g(x_l, y, s)) \tag{5.1}$$

Algorithm 1 Parallel stereo-matching estimates the disparity *s* for pixel (x_l, y_l) in the left image.

Require: the pixel (x_l, y_l) in the left image and those candidate pixels (x_r, y_r) in the right image.

Ensure: the disparity *s*

```
for x_r ← 0 To x_l;  y_r = y_l do
    SSD;          // computing the window aggregated matching cost SSD for pixel (x_l, y_l) and (x_r, y_r);
    if SSD == MIN then
        s ← x_l − x_r;
    end if
end for
Return s;
```

Note that the left image pixel is used as the reference in the cost computation as default. So, we use the l subscript in the estimated disparity symbol, *s* ranging from s_{min} to s_{max}.

5.4/ APPLICATION TO BALANCED STRUCTURED MESHING

In this section, we first present the standard on-line and sequential self-organizing map algorithm as stated originally by Kohonen [Koh01, Koh82]. We present the problem of structured mesh generation as a balanced clustering problem in the plane dealing with an hexagonal topological grid (the neural network grid or mesh) that must adapt to an underlying distribution in the plane (the distribution map). Then, we detail the parallel model proposed for the SOM implantation on GPU and its application to both balanced stuctured mesh problem and also traveling salesman problem.

5.4.1/ THE SELF-ORGANIZING MAP ALGORITHM

The standard self-organizing map [Koh01, Koh82] is a non directed graph $G = (V, E)$, called the network, or topological grid, or structured mesh, where each vertex $v \in V$ is a neuron having a synaptic weight vector $w_v = (x, y) \in \Re^2$, $\Re^2$ 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 $d(v, v')$.

Algorithm 2 Sequential self-organizing map training procedure.

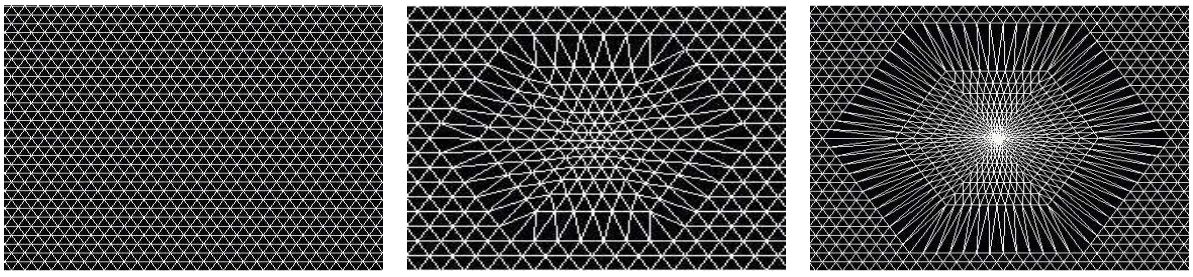
```
Randomly generate a network of neurons (regular grid or ring);
for iter ← 0 To t_max do
    Randomly extract a point p from the data set;
    Perform competition to select the winner neuron n* according to p;
    Apply learning law to move the neurons of a neighborhood of n*;
    Slightly decrease learning rate α and radius σ of neighborhood
end for
```

The training procedure structure is given in Algorithm 2. A fixed amount of t_{max} iterations are applied to a network, the vertex coordinates of which being initialized to a regular

hexagonal grid (each vertex with 6 neighbors), or a randomly generated ring (for TSP). The data set is either a distribution map or a set of cities, from which are extracted data points by a roulette wheel mechanism. Note that the distribution map (disparity map) stands for a density map, where pixel value represent some point density value in the plane. 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). 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 5.2 is applied to n^* and to the neurons within a finite neighborhood of n^* of 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 the Gaussian in Equation 5.3. Here, 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, coefficients $\alpha(t)$ and σ_t are multiplied by $e^{ln\frac{\chi_{final}/\chi_{init}}{t_{max}}}$ with $\chi = \alpha$ and $\chi = \sigma$, χ_{init} and χ_{final} being, respectively, the values at starting and final iteration. Note that a SOM simulation is characterized by the five running parameters $(\alpha_{init}, \alpha_{final}, \sigma_{init}, \sigma_{final}, t_{max})$.

$$w_n(t+1) = w_n(t) + \alpha(t) \times h_t(n^*, n) \times (p(t) - w_n(t)) \tag{5.2}$$

$$h_t(n^*, n) = e^{\frac{-d_G(n^*,n)^2}{\sigma_t^2}} \tag{5.3}$$

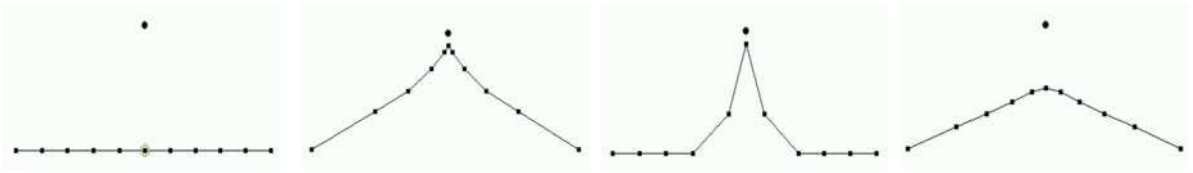


(a) (b) (c)

Figure 5.3: A single SOM iteration with learning rate α and radius σ. (a) Initial configuration. (b) $\alpha = 0.5, \sigma = 6$. (c) $\alpha = 1, \sigma = 12$.

Application of SOM to the stereo disparity map consists of applying the training procedure to a structured hexagonal mesh (the network). According to a density map, that is a simple transformation of the disparity map. Examples of a basic iteration with different learning rates and neighborhood sizes are shown in Fig. 5.3 for such a hexagonal regular grid, where each vertex has exactly 6 neighbors. The density map used for training is a simple transformation of the disparity map. It consists, before the training starts, in removing background values (very low disparity values) and increasing contrast between disparity values. This is done in order to increase the data point density for objects that are closest to the camera in the scene. Here, density values are set to the square of the disparity values.

In the TSP application, the grid which is used for representing the traveling salesman tour is a ring structure with a fixed number of vertices (neurons) M. Specifically, M is set to



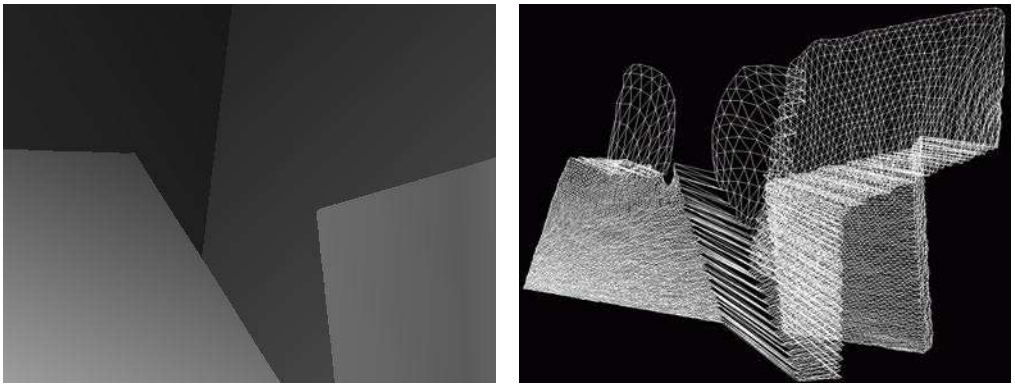
(a) (b) (c) (d)

Figure 5.4: A single SOM iteration with learning rate α and radius σ. (a) Initial configuration. (b) $\alpha = 0.9, \sigma = 4$. (c) $\alpha = 0.9, \sigma = 1$. (d) $\alpha = 0.5, \sigma = 4$.

$2N$, N being the number of cities. Examples of a basic iteration with different learning rates and neighborhood sizes are shown in Fig. 5.4.

5.4.2/ THE BALANCED STRUCTURED MESH PROBLEM

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 left image and the right image obtained by a stereo camera. The disparity map is a 2D image which represents the 3D surface "seen" by a camera. But few approaches of stereo-matching and surface reconstruction currently match the requirements of real-time execution. Algorithms on GPU are now developed to respond to this problematic. Furthermore, the volume of data stored and manipulated in the disparity map is often very important and may constitute an obstacle for real-time algorithms. Here, we address the problem of building, in a massively parallel way and by using GPU, a compressed and adapted structured mesh representing a given disparity map. The goal is to represent the 3D scene in a compressed and efficient way, with more details for objects that are close to the camera and less for objects that are far from it. The compressed structured mesh obtained in 2D space, as it is the disparity map, should then allow for fast treatment or visualization in 3D space, as illustrated in Fig. 5.5.



(a) (b)

Figure 5.5: 3D reconstruction example (a) Input disparity map in 2D. (b) The compressed grid in 3D space.

Here, the Kohonen's [Koh01, Koh82] self-organizing map (SOM) algorithm is used as an heuristic, to achieve such a goal of structured compressed mesh generation according to a disparity map in a massively parallel way on GPU. The disparity map is the input. It is used as a 2D density distribution on which the algorithm operates in a massive parallel fashion by deformation of a 2D hexagonal grid, called the neural grid. The neural grid can be seen as a visual pattern that adapts and modifies its shape according to the underlying distribution. The important property is that the grid deformation can respect the density distribution and topology. This means that high disparity values will be represented by higher neural grid point densities and that grid points hexagonal neighborhoods will reflect spatial topology, or distances in 2D and 3D space.

In order to evaluate the adequacy of the generated grid according to the underlying density map, we state the problem as a balanced clustering problem in the plane, called balanced structured mesh problem, where nodes define deformable cells that cover some constant part of data density. The definition of the balanced structured mesh optimization problem needs to consider both the hexagonal grid (the mesh) and the underlying density distribution map. Fig. 5.6 illustrates the main elements. The target adaptive hexagonal mesh is illustrated in the left part of the figure. 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 points they cover. The right part of the figure shows such an hexagonal cell and its covering points from the density map. In the example of the figure, the total value covered, called the weight of the honeycomb cell, is the summation of the underlying point values.

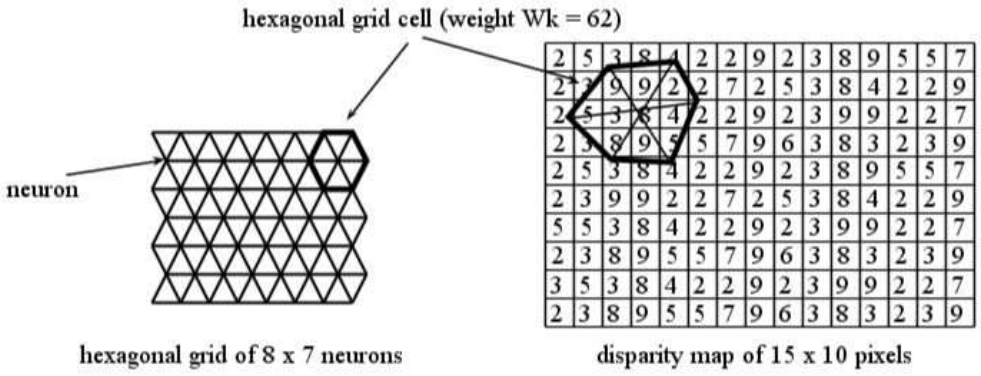


Figure 5.6: Hexagonal structured mesh with honeycomb cells and triangular subdivision (left), density map covering with of a single honeycomb cell with weight $W_k = 62$.

Let M_k be the set of honeycomb cells of the mesh. Let W_k be the weight of a single honeycomb cell, defined as the sum of the underlying point values from the distribution map. Note, that this weight can be computed by using a standard pixel coloring algorithm applied to the input distribution map.

$$W = \frac{\sum_k W_k}{K} \tag{5.4}$$

$$Cost = 100 \times \frac{\sum_k |W_k - W|}{K \times W} \tag{5.5}$$

Let W be the average weight of the K honeycomb grid cells defined by Equation 5.4. We define the optimization problem as the minimization of the average percentage de-

viation of each individual honeycomb cell weight to the average honeycomb cell weight as defined in Equation 5.5. Hence, the structured mesh generation problem consists in minimizing this criteria while preserving the regularity of hexagonal topology, such geometrical constraints being only visually verified in this document.

5.4.3/ EUCLIDEAN TRAVELING SALESMAN PROBLEM

We also use the SOM as an heuristic to address the Euclidean traveling salesman problem (TSP). In that case, the network is a ring randomly initialized. The ring will deploy from scratch and progressively will match the cities.

The general traveling salesman problem can be simply defined as a complete weighted graph $G = (V, E, d)$, where $V = \{1, 2, \cdots, N\}$ is a set of vertices (cities), $E = \{(i, j)|(i, j) \in V \times V\}$ is a set of edges, and d is a function assigning a weight d_{ij} to every edge (i, j). The objective is to find a minimum weight cycle in G which visits each vertex exactly once. The Euclidean TSP, or planar TSP, is the TSP where cities are points of the plane and weight is the Euclidean distance between cities. It consists, correspondingly, of finding the shortest tour that visits N cities where the cities are points in the plane and where the distance between cities is given by the Euclidean metric.

5.4.4/ PARALLEL CELLULAR MODEL

The goal of structured mesh generation is to homogeneously divide a density map, the input distribution map in our applications, between the many triangles of the structured hexagonal mesh. In the TSP, the goal is to deploy a ring network among the cities and generate a smallest tour. We explain here the main components and algorithmic procedures of the cellular parallel model for SOM and these two applications. First, we present the data treatment partition according to the cellular matrix. Secondly, we detail the data impact on the thread activity which depends on data density. Third, we explain how parallel spiral search are performed for closest point findings, and how this procedure is critical according to computation time performance. Then, we illustrate two results of the two applications by visual representation.

5.4.4.1/ DATA TREATMENT PARTITION

We first present the model for structured meshing and below its adaptation for TSP. For the balanced meshing problem, the space is defined by the distribution map dimensions of size $W \times H$, and each grid point can move on the density map. Given an input data density map of size $W \times H$, a two-dimensional topological grid is created, of a given size $W_g \times H_g$. Note that each grid node is indexed in its grid position, and that the size of the grid is lower than the size of the density map, in such a way that the grid constitutes a compressed representation of it. Note also that each grid node has coordinates (neuron weights) in the Euclidean plane, and that these coordinates are defined by the dimensions of the density map.

In order to implement the parallel level, at which parallel execution will take place, we now introduce a supplementary level of decomposition of the plane and input data: the cellular matrix. Between the topological grid and the density map, we now introduce a

two-dimensional cellular matrix of size $W/co \times H/co$, where co is a preset constant factor. The three main data structures of the parallel model are illustrated in Fig. 5.7. This intermediate cellular matrix is in linear relationship to the input size. To each cell corresponds a single thread. Its role is to memorize the grid nodes in a distributed fashion and authorize many parallel closest point searches in the plane by a spiral search algorithm [BWY79].

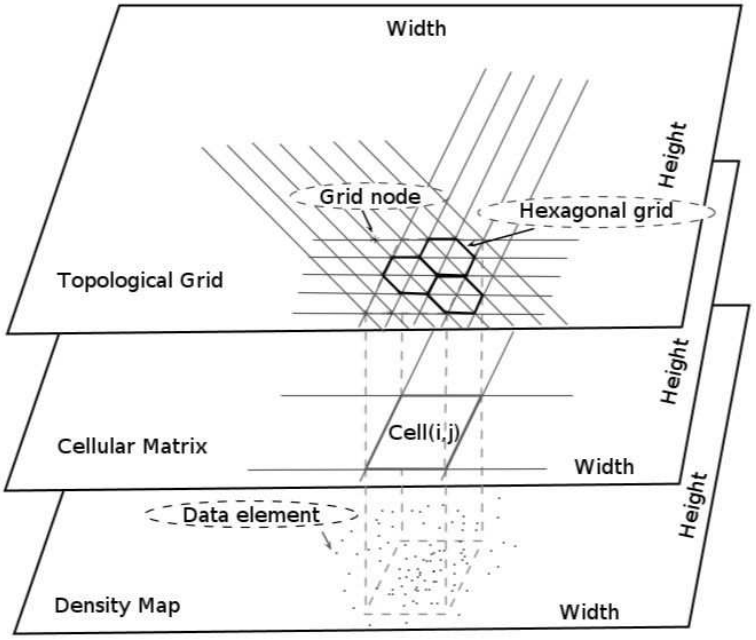


Figure 5.7: Parallel cellular model: the input density map, the cellular matrix and hexagonal grid. To a given cell corresponds a part of the input data, a part of the structured mesh, and a thread.

Each cell in the cellular matrix is a basic training unit and will be handled by one thread. Each cell is responsible for executing a complete SOM process locally. It is at this level that massive parallelism takes place since the threads correspond one-to-one with the cells and are in linear correspondence to the input density map size, in $O(W \times H)$.

Application to the traveling salesman problem is very similar. The three main data structures of the parallel model are illustrated in Fig. 5.8. The input distribution map is the set of cities in the plane itself. The network is now a ring of size $M = 2N$, where N is the number of cities. The scale of cell partition is $\lceil \sqrt{N \times \lambda} \rceil^2$, with λ a constant factor, set to $\lambda = 1.1$ in experiments. The number of parallel processors needed is $O(N)$. The memory complexity is also $O(N)$.

The many SOM processes that are activated simultaneously correspond to the cellular matrix decomposition. The four steps of the SOM algorithm are executed in a distributed way. These steps are the input data point extraction step according to the density map, the closest point or winner neuron finding step, the application of learning rule step, and the parameter decrease step. Then, this process, performed independently in parallel by the many cells/threads, 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. We now give more details of each SOM main steps. Algorithm 3 resumes the parallelized SOM algorithm with cell partition. An important point for the algorithm to be correct, is to guarantee a parallel data

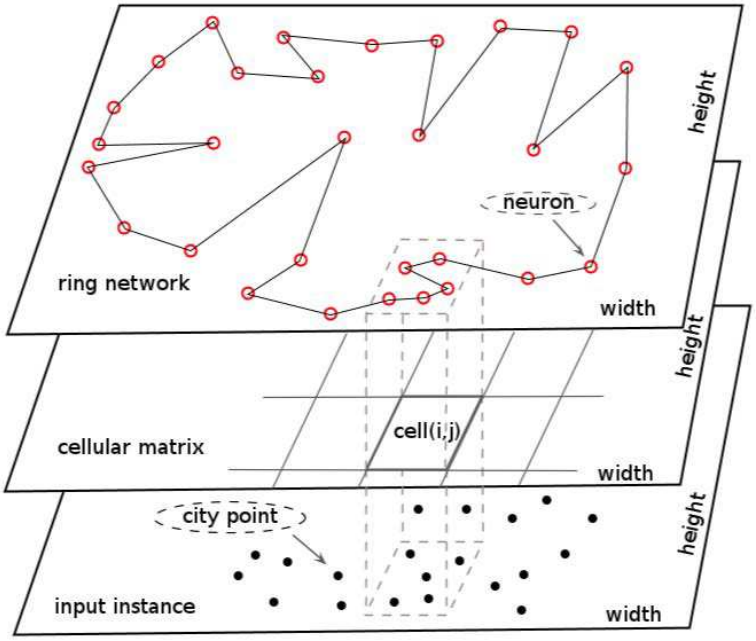


Figure 5.8: Parallel cellular model: the input cities, the cellular matrix and the ring structure, used in TSP implementation. To a given cell corresponds a part of the input data, a part of the ring, and a thread.

extraction step that reflects the data distribution. We detail that point.

Algorithm 3 Parallel self-organizing map training procedure.

```
1: Generate a regular grid of neurons;
2: Calculate activation probability for each cell;
3: for iter ← 0 To t_max do
4:    Check in parallel whether cell_i(i = 1, 2, ..., num) is activated or not;
5:    if cell_i is activated then
6:       Randomly extract data point (from density map or cities) p from the cell_i;
7:       Perform competition to select the winner neuron n* according to p;
8:       Apply learning rule to move the neurons of a neighborhood of n*;
9:       Wait until all the other cells finish their works;
10:      Slightly decrease learning rate α and radius σ of neighborhood
11:   end if
12: end for
```

5.4.4.2/ THREAD ACTIVATION AND DATA POINT EXTRACTION

$$p_i = \left(\frac{q_i}{\max\{q_1, q_2, \ldots, q_{num}\}}\right) \times \delta \tag{5.6}$$

First, is the problem of random data point extraction by a roulette wheel mechanism performed in parallel and according to the density distribution. As a solution to this problem, we propose a particular cell activation formulae in Equation 5.6 to determine if a cell/thread will be activated at each parallel iteration. The p_i in the equation is the probability

that the cell *i* will be activated. Here, the references of the parameters in the equation are different following different problems:

- for balanced meshing processing, q_i is the sum of the pixel values of the density map that correspond to the cell *i*, i.e. the density of the cell, and *num* is the total number of cells. Then, a cell is activated proportionally to its density value according to the whole density map. Hence, the cell with the highest density will be activated with probability one. The empirical preset parameter δ is used to adjust the level of activity of the threads when necessary in order to avoid too many memory access conflicts. In this application, parameter δ will be set to 1. The influence of δ in the iteration is shown in Fig. 5.9 for a single step parallel iteration.
- for the TSP processing, q_i is the number of cities into the cell *i*, and *num* is the number of cells. The empirical preset parameter δ is used to adjust the degree of activity. As a result, the more cities a cell contains, the higher is the probability this cell to be activated to carry out the SOM execution at each parallel iteration. In this way, cell activation depends on the input data density distribution.

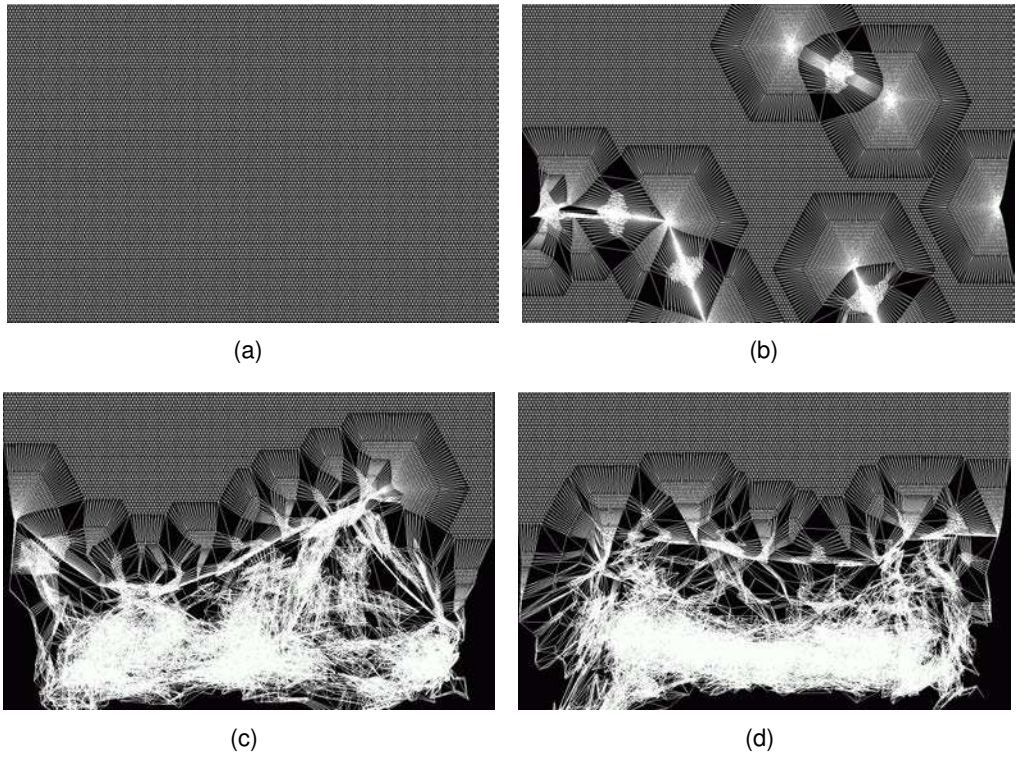


Figure 5.9: Influences of different δ values on the "cones" image pair. (a) initial state. (b) $\delta = 0.1$. (c) $\delta = 0.5$. (d) $\delta = 1.0$.

$$prob(i) = \frac{d_i}{\sum_{j=0}^{cellNum} d_j} \tag{5.7}$$

To complete the extraction step, and obtain a data point in the plane, the added strategy could vary:

- for the disparity map meshing processing, each activated cell/thread simply performs a local roulette wheel mechanism into the cell itself, in order to produce the extracted pixel in proportion to its local intensity. The probability of a pixel choice local to a cell is defined by Equation 5.7, where d_i the density value of that pixel in the cell, *cellNum* is the number of pixels in the cell, and d_j their density values.
- for the TSP processing, the activated processing unit randomly chooses a city from the cities located in its own cell with uniform probability, unlike the original sequential SOM which randomly extracts a point from the entire input data set.

5.4.4.3/ PARALLEL SPIRAL SEARCH AND LEARNING STEP

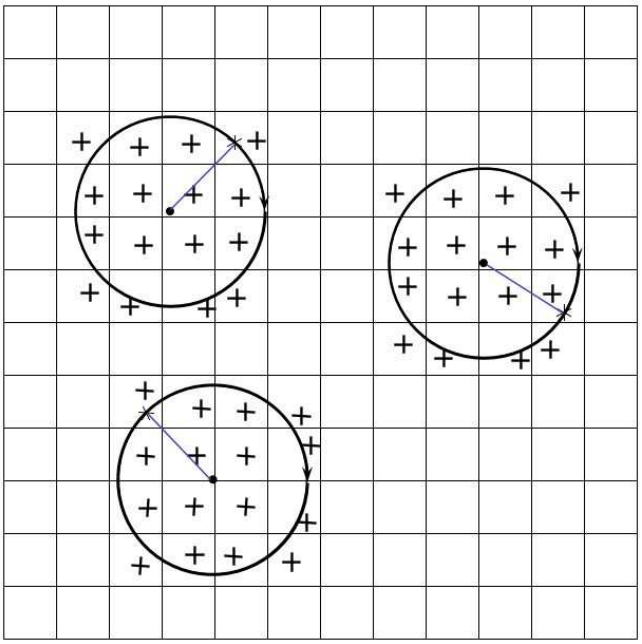
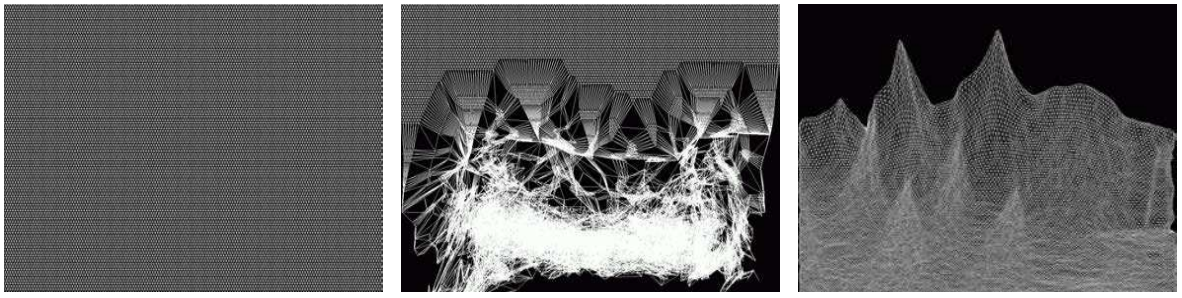


Figure 5.10: Spiral searches performed in parallel.

After the extraction step, each activated GPU thread has gotten its input data point for further operations. The next job is to find out the nearest topological grid node corresponding to the extracted point. Each activated GPU thread, with its input extracted point, performs a spiral search [BWY79] on the cellular matrix that contains the topological grid nodes. Spiral searches operate in parallel as illustrated in Fig. 5.10. The spiral search proceeds from the cell that contains the extracted point, as in the middle of the circle of the figure and progressively extends the search to the cells surrounding until a grid node is found. Once there is one grid node that is found, it is guaranteed that it is not necessary to search any cell, that does not intersect the circle of radius equal to the distance to the first grid node found and centered at the extracted point. 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. 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, or the number of cities, thus transforming a $O(N)$ sequential algorithm search into a constant time $O(1)$ theoretical parallel algorithm in the average case for bounded distributions.

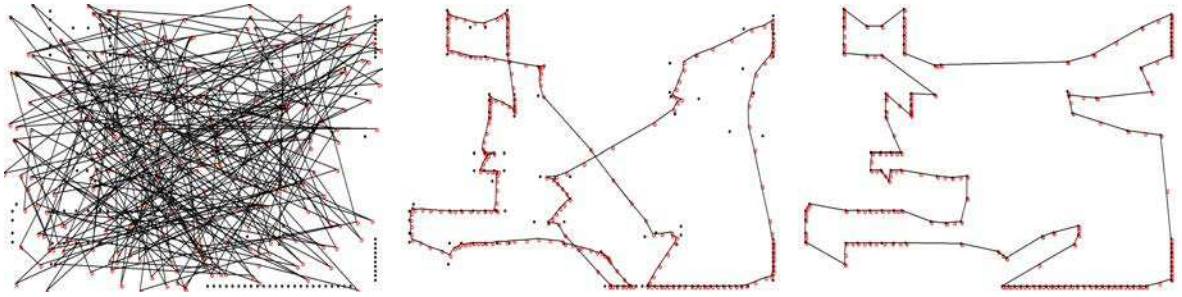
The next step is the learning step, each thread performs the displacements of the grid nodes in the plane according to the learning rule. The displacement involves not only the grid node found (winner neuron), but also its closest neighbors in the sense of the topological distance into a neighborhood of radius σ_t. The modifications of the grid node locations is done according to Equation 5.2 and Equation 5.3. The last step in SOM processing is to decrease the learning parameters. The main operation is to modify the values of the learning intensity and the radius of neighborhood. In experiments, this specific last step is carried out on CPU for convenience in order to avoid the load and duplication of the parameters into the GPU memory space.

5.4.4.4/ EXAMPLES OF EXECUTION



(a) (b) (c)

Figure 5.11: Different steps of training procedure on the “cones” disparity map instance.



(a) (b) (c)

Figure 5.12: Different steps of training procedure on the $pr124$ instance.

The main parallel algorithm repeats t_{max} times the parallel iteration. To resume the algorithm behavior, Fig. 5.11 and Fig. 5.12 present visual representations of the solutions at different iteration steps of the simulation, for the two applications respectively. Fig. 5.11 presents meshing on the “cones” image pair from Middlebury database and presented in Appendix. Fig. 5.11(a) presents the initial regular grid. Fig. 5.12 presents application to TSP instance $pr124$ from TSPLIB [Rei91]. Fig. 5.12(a) presents the random initial ring for TSP. For each application case, the other two figures (b-c) present the simulation at intermediate step, and final step respectively. At the end of the iterations, the hexagonal

grid can represent the ‘texture’ of the input density map in Fig. 5.11(c). Similarly, at the end of the process, the ring network has almost completely been mapped onto cities, as illustrated in Fig.5.12(c).

5.5/ CONCLUSION

In this chapter, we have defined the proposed cellular GPU parallel model, which is the central heart of our work. We figured out the most important step in parallel model, the partitioning of the input distribution map. The cellular model has adapted a cellular matrix into the execution. The small tasks from the partitioned input distribution map are assigned to cells of the cellular matrix, and the cells are assigned to processing units (GPU threads). Resulting from these partitioning and mapping of tasks, the needed cores for computation is linear with the size of input distribution density map, which brings the main advantage of the proposed cellular parallel model: dealing with large size problems. Then, we detailed the cellular parallel model for applications we have chosen for testing the efficiency of the model. The applications are in two groups: the stereo-matching problems and the balanced meshing processing, with also application to the standard Euclidean traveling salesman problem. Experiments based on this model will be introduced in following chapters.

6

GPU IMPLEMENTATION OF CELLULAR STEREO-MATCHING AGORITHMS

6.1/ INTRODUCTION

In last chapter, we have presented stereo-matching problems as good candidates for parallel GPU processing. We have presented a general cellular parallel model intended to handle general Euclidean problems by data partition, and where local dense stereo-matching method naturally fits. In this chapter, we present the details of two GPU stereo-matching methods. The first application presents the main algorithmic components necessary for stereovision on CFA image pairs. The basic building blocks of a standard GPU local dense stereo-matching implementation are outlined and discussed. Starting from such preliminary study, different mechanisms are proposed to enhance performances on both solution quality and computation time. They are exposed in the second application devoted to GPU real-time stereovision implementation. We present the modifications at the levels of memory management, cost computation, adaptive window aggregation and disparity map refinement step. This new approach will lead to speedup in computation time and quality. Results and comparative evaluations are illustrated by experiments on standard image processing databases. Evaluations are systematically performed according to the image size increase.

6.2/ CFA DEMOSAICING STEREOVISION PROBLEM

The method that is presented in this section is an alternative solution to match pixels by analyzing the CFA images without reconstructing the full color image by demosaicing processing. The following contents in this section are organized in two parts. We first present the partial demosaicing matching method including the estimation of the second color component and the matching cost. Then, we describe the implementation on CUDA and the experiments to evaluate the performances.

6.2.1/ CFA DEMOSAICING STEREOVISION SOLUTION

6.2.1.1/ CFA STEREOVISION PROCESS

For better carrying out the comparison, the procedures of the experiments on CPU and on GPU are almost the same. The total processing can be illustrated as Fig. 6.1.

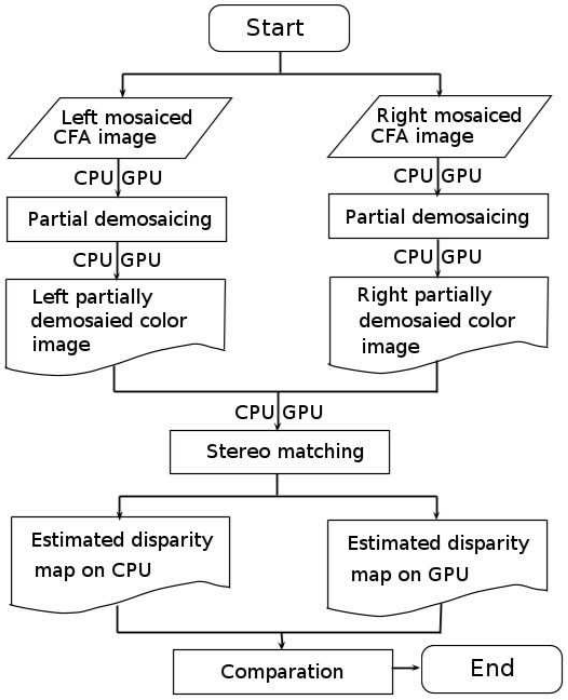
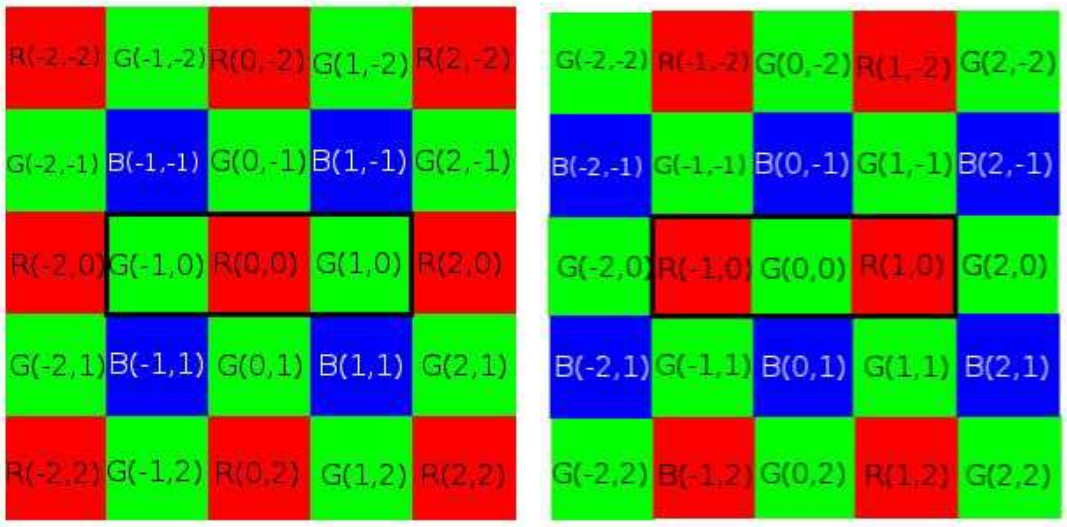


Figure 6.1: Partial demosaicing stereo-matching method flowchart.

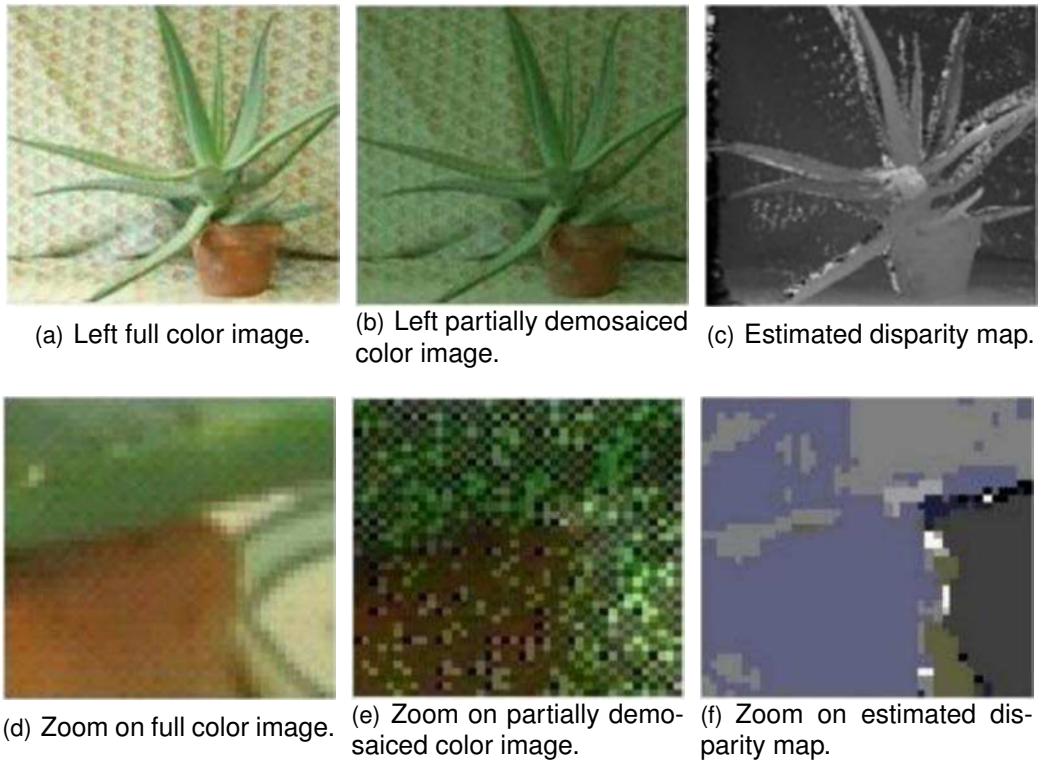
Since the demosaicing methods intend to produce demosaiced color images, which are maybe ”visually pleasing”, they will try to reduce the presence of color artifacts, such as those false colors and zipper effects, by filtering the images [YLD07]. That may sometimes lead to the rejection of color texture information, which is useful to match homologous pixels, from the demosaiced color images, rather than displaying images. The straightforward stereo-matching methods, matching pixels in image pairs acquired with single-CCD cameras, are to first obtain the CFA images. As the datasets are all full color images, at the very beginning of the experiments, a simulation step for every pair of images is realized to obtain their CFA images needed, by keeping only one of the three color components at every pixel. This work is done by GPU and by CPU, respectively. The whole evaluation is performed according to the spatial arrangement of the Bayers CFA, referring to Fig. 6.2. Then, for every CFA image, the partial demosaicing step is carried out, on GPU and on CPU, respectively. So, the left demosaiced color image and the right one are produced. The estimation method is the edge-adaptive demosaicing method proposed by Hamilton *et al* as presented in [HA97].

The original full color image, as shown in Fig. 6.3(a), and the left demosaiced color image by Hamilton’s method, as shown in Fig. 6.3(b), look somewhat similar. However, zooming on the square areas outlined in these images, as shown in Fig. 6.3(d) and Fig. 6.3(e), shows that textured areas are locally different.



(a) Original color component. (b) The SCC to estimate.

Figure 6.2: CFA color components and their SCC.



(a) Left full color image. (b) Left partially demosaiced color image. (c) Estimated disparity map.

(d) Zoom on full color image. (e) Zoom on partially demosaiced color image. (f) Zoom on estimated disparity map.

Figure 6.3: 'Aloe' left image.

At the end, a disparity map is estimated for each pair of images, as shown in Fig. 6.3(c).

6.2.1.2/ SECOND COLOR COMPONENT

The partial demosaicing matching method starts with the mosaiced CFA images. Here, the CFA images are those images obtained according to the Bayer color filter. Each two-by-two submosaic contains 2 green, 1 blue and 1 red filter. Each submosaic covering one pixel sensor. The mosaiced CFA image is the one whose pixel contains only one color component according to the Bayer color filter.

Different from the classical methods, which estimate all the missing color components for every pixel in the CFA images, the partial demosaicing method estimates only one color component, the Second Color Component (SCC), for every pixel. Here, the SCC is defined as the color component that is available in the same line, as illustrated in Fig. 6.2. This means that SCC is the green color for all the red and blue pixels, while for the green pixels the SCC is the red color component for even lines and the blue color component for odd lines, as in Equation (6.1).

$$SCC(x,y) = \begin{cases} \hat{G}(x,y) & \text{for red and blue pixels} \\ \hat{R}(x,y) & \text{for green pixels in even lines} \\ \hat{B}(x,y) & \text{for green pixels in odd lines} \end{cases} \tag{6.1}$$

6.2.1.3/ SCC ESTIMATION

This method is an edge-adapted demosaicing method presented by Hamilton and Adams[HA97]. To select the interpolation direction, this method takes into account both gradient and Laplacian second-order values, by using the green levels available at nearby pixels and red (or blue) samples located two apart.

For example, in the case of GRG, as illustrated in Fig. 6.2(a), to estimate the missing green level at the red pixels. This method uses the following algorithm:

a) Approximate the horizontal Δ^x and vertical Δ^y gradients thanks to absolute differences as Equation (6.2).

b) Interpolate the green level as Equation (6.3).

$$\begin{cases} \Delta^x = \left|G_{-1,0} - G_{1,0}\right| + \left|2R - R_{-2,0} - R_{2,0}\right| \\ \Delta^y = \left|G_{0,-1} - G_{0,1}\right| + \left|2R - R_{0,-2} - R_{0,2}\right| \end{cases} \tag{6.2}$$

$$\hat{G} = \begin{cases} \frac{G_{-1,0}+G_{1,0}}{2} + \frac{2R-R_{-2,0}-R_{2,0}}{4} & \text{if } \Delta^x < \Delta^y \\ \frac{G_{0,-1}+G_{0,1}}{2} + \frac{2R-R_{0,-2}-R_{0,2}}{4} & \text{if } \Delta^x > \Delta^y \\ \frac{G_{-1,0}+G_{1,0}+G_{0,-1}+G_{0,1}}{4} + \frac{4R-R_{-2,0}-R_{2,0}-R_{0,-2}-R_{0,2}}{8} & \text{if } \Delta^x = \Delta^y \end{cases} \tag{6.3}$$

Since this method well combines two color components in partial derivate approximations by exploiting spectral correlation in the green plane estimation, the precision is well guaranteed.

Each pixel with coordinates (x, y) in the partially demosaiced color images is characterized by a two-dimensional partial color denoted $\hat{I}_{PA}$. As shown in Equation (6.4), this partial color point is composed of the available color component and the estimated second color component.

$$\hat{I}_{PA}(x,y) = \begin{cases} (R(x,y), \hat{G}(x,y))^T & \text{if x is odd and y is even} \\ (\hat{R}(x,y), G(x,y))^T & \text{if x is even and y is even} \\ (\hat{G}(x,y), B(x,y))^T & \text{if x is even and y is odd} \\ (G(x,y), \hat{B}(x,y))^T & \text{if x is odd and y is odd} \end{cases} \tag{6.4}$$

6.2.1.4/ MATCHING COST

The used matching method is a local dense stereo-matching method also called window-based approach. It respects the very assumption that the intensity levels of neighbors of a left pixel are close to those of the same neighbors of its homologous right pixel in the right image. So, the matching cost is defined between the window around the left pixel and the window around the candidate right pixels in the corresponding line (epipolar line) in the right image. The window is shifted over all possible pixels so that a matching cost between the left pixel and each candidate in the right image is obtained. By the Winner-Takes-All method, the final disparity estimation is realized by selecting the shift with the lowest matching cost.

The matching cost, Sum of Squared Differences cost (SSD), is adapted as Equation (6.5), based on Equation 3.8.

$$SSD(x_l, y, s) = \sum_{i=-\omega}^{\omega} \sum_{j=-\omega}^{\omega} \| \hat{I}_{lPA}(x_l + i, y + j) - \hat{I}_{rPA}(x_l + i - s, y + j)\|^2 \tag{6.5}$$

Where $\| \bullet \|$ is the Euclidean norm, while s is the spatial shift along the horizontal epipolar line and ω the half-width of the $(2\omega + 1) \times (2\omega + 1)$ aggregation window.

Since these pixels of horizontal lines with the same parity in the left and right partially demosaiced color images are characterized by the same two color components, we can reasonably assume that the partial color points of two homologous pixels are similar. Because the partial costs compare the partial color points of left and right pixels located on the same horizontal lines, they reach an extremum when the shift is equal to the disparity.

6.2.2/ EXPERIMENT

6.2.2.1/ EXPERIMENT PLATFORM

We use the well known Middlebury databases [HS06]. The ten datasets used in our experiments are entitled *'Aloe', 'Bowling1', 'Cloth1', 'Flowerpots', 'Lampshade1', 'Midd1',*

'Monopoly', 'Plastic', 'Rocks1' and *'Wood1'*. All the datasets of 2 views are used here. The dimensions of the images are different (measured in pixels): the full size images with height of 1110 and width from 1240 to 1396, the half size images with height of 555 and width from 620 to 689, the small size images with height of 370 and the width from 413 to 465.

The datasets used are illustrated in Appendix A in Fig. A.1, Fig. A.2 and Fig. A.3, respectively. In these datasets, the color stereo images are acquired by high resolution cameras equipped with one-single-sensor [SS02], that's to say, the full color images are in fact color images that have been demosaiced by a specific chip integrated in the camera. They could contain artifacts caused by the demosaiced step. What's more, the work of applying a demosaicing step on CFA images, which have been generated by sampling color components from these previously demosaiced color images, involves applying two successive demosaicing steps on the CFA images acquired by the camera. However, since Middlebury is the most frequently utilized databases, a comparison of performance on different platform will be quite indispensable.

Our experiments are carried out both on CPU and GPU. For the CPU, it is an Intel(R) Core(TM)2 Duo CPU E8400 of 3.00GHz, and each of the two cores with a cache of 6144KB. The GPU used in the experiments is a GPU GeForce GTX 570 of NVIDIA.

The system, on which the experiment is evaluated, is Ubuntu 11.04 of 32 bits. The programming interface we used for parallel computation on GPU is Compute Unified Device Architecture (CUDA).

6.2.2.2/ CUDA IMPLEMENTATION

Based on our experiment platform, we briefly lay out the CUDA settings and the parameter values mentioned in our algorithm, taking an image with the size of $W \times H$ as an example.

In our experiments, we use the two-dimension *block* of size 16×16. Each thread takes care of one pixel and the size of *grid* is obtained by $\frac{W+16-1}{16} \times \frac{H+16-1}{16}$ for a given $W \times H$ image. In this experiment, we have created two *grids*. The first one is for the SCC estimation step, a *grid* of $W \times H$ threads is created and each thread takes care of one pixel in the estimation. For the matching cost computation, a second *grid* of $W \times H$ threads is employed, to compute the matching cost for every pixel at a set of given disparities and then pick out the best homologous candidate pixel by the Winner-Takes-All method.

In the two steps, the estimation of SCC and the stereo-matching, the executions are carried on CPU and GPU, respectively. On CPU, for these two steps, all the works are organized serially. We stock all the date in linear by line priority and we estimate the SCC one-by-one for all the pixels horizontally. The matching is done in the same way. On GPU, though the data is stocked in the same way, the executions of jobs are different, referring to Fig 6.4. The *kernels* carried out by multi-cores will be sent to GPU, as illustrated in Fig 6.5 (in the figure, the *width* and the *height* are the dimensions of images). Every thread takes in charge of one partition for the estimation of SCC and for the stereo-matching illustrated in Algorithm 4. The granularity of each execution is determined by the number of threads per block. Since these threads are executed in parallel, the computation is accelerated.

A matching is considered as valid when the absolute difference between the estimated disparity and the given benchmark $d_l^{\omega}(x_l, y)$ is lower or equal to δ, which is the disparity

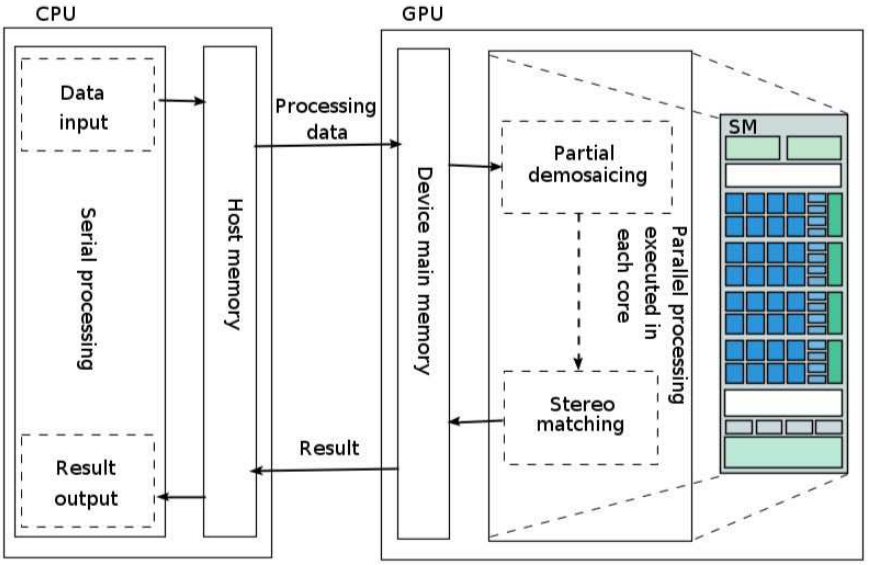


Figure 6.4: CUDA processing flow on CPU and GPU in the experiments.

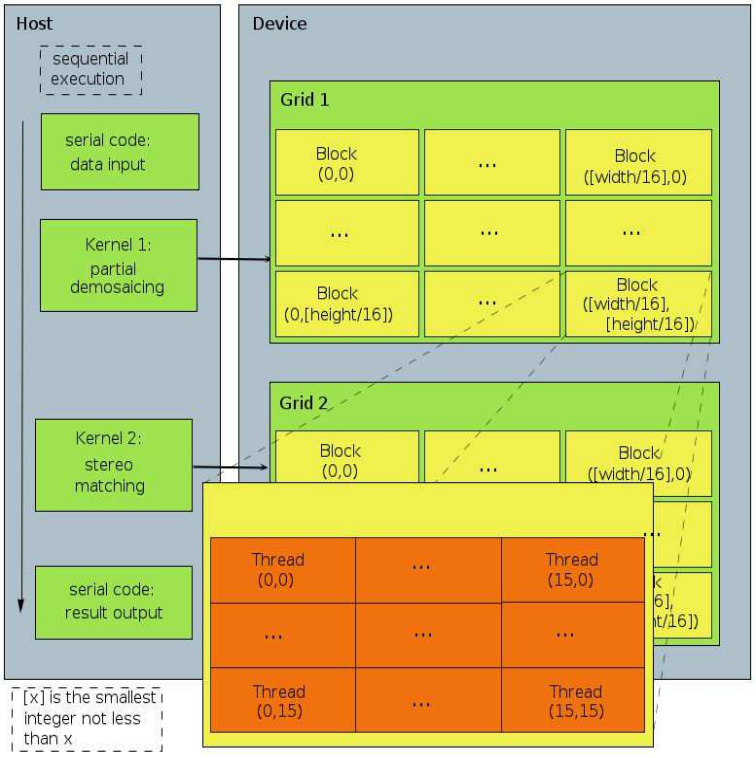


Figure 6.5: CUDA model in the experiments.

Algorithm 4 Executions on GPU.

```
1: Specify dimensions of grids and blocks;
2: Allocate the problem data input on GPU device memory;
3: Allocate SCC-arrays and CFA-arrays on GPU device memory;
4: Copy the problem data input on GPU device memory;
5: Kernel-1<<< ... >>> (...)          // estimating the SCC for every pixel;
6: Free problem inputs;
7: Allocate estimated-disparity-array on GPU device memory;
8: Kernel-2<<< ... >>> (...)          // stereo-matching by algorithm 1 for every pixel;
9: Free SCC-arrays and CFA-arrays;
10: Copy the estimated-disparity-array on CPU host memory;
11: Free estimated-disparity-array;
```

error tolerance. In the experiments this coefficient is set to 1.

6.2.2.3/ EXPERIMENT RESULTS

The experiments are executed on all the ten chosen datasets. The experiment results are shown in Appendix B in Table B.1 for full size images, Table B.2 for half size images and Table B.3 for small size images, respectively.

Here, we take the 'Aloe' image pair as an example. With the increase of the half-window, the computation time increases significantly. This results from the increase of the computing density. When the half-window is given as ω, for every pixel in the left image and for their every possible candidate pixel in the right image, we should compute with a set of $(2\omega + 1) \times (2\omega + 1)$ pixels to obtain the matching cost. So, when the half-window ω increases, the computing complexity is a squared function of ω, which is a real challenge to the capacity of the processors.

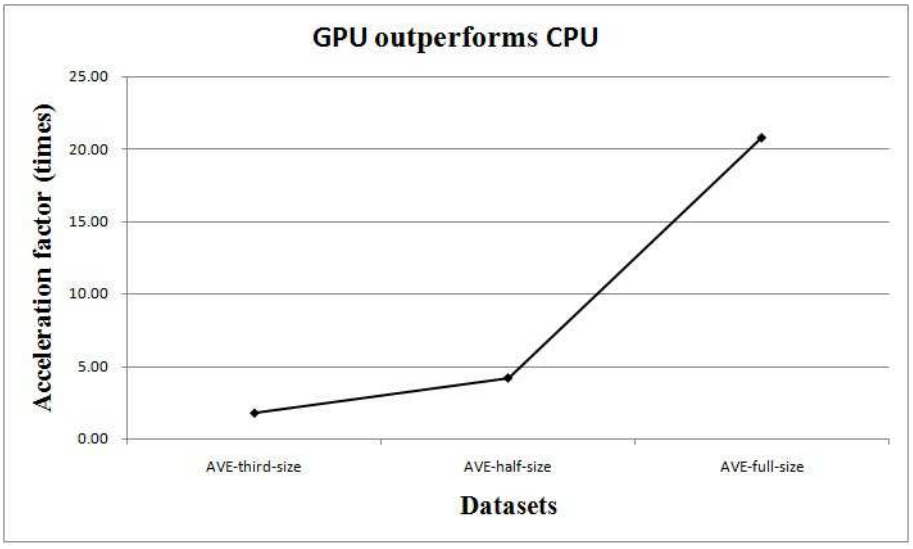


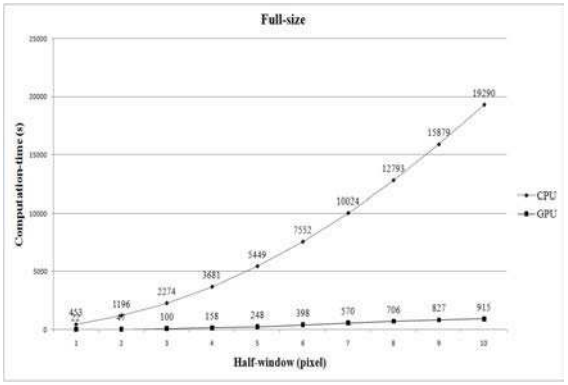
Figure 6.6: GPU outperforms CPU in terms of computation time. The average acceleration obtained in the executions on 'Aloe' image pair based on half-window from 1 to 10 is marked. The acceleration increases along with the image dimensions.

Meanwhile, with the increase of the half-window, and so, the computation intensity, CPU's computation time increases more importantly than GPU's computation time. Moreover, as illustrated in Fig. 6.6, as the image dimensions augment, the acceleration factor obtained by using GPU has expanded from 1.75 to 20.75. This means that the GPU offers powerful computation capacity in intense computation thanks to the parallel organization of the *blocks* and the *threads* on GPU.

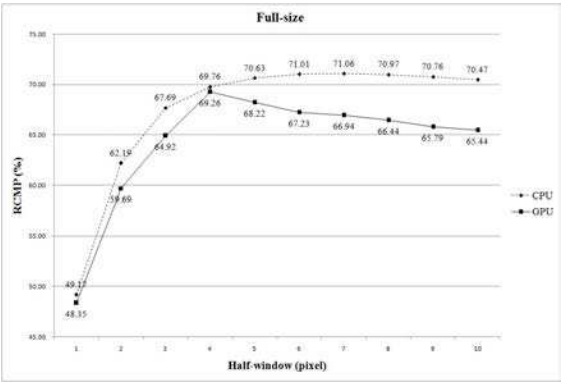
The performance of GPU is significantly influenced by the absolute dimension of the chosen image; here the absolute dimension of an image is defined as the sum of pixels in the image by multiplying width and height. The nearer to the complete load of the employed streaming multiprocessor on GPU, the higher performance we get.

The Ratio of Correctly Matched Pixel (RCMP) is the percentage of the well matched pixels in all the pixels of the image to be matched. As it is shown in Fig. 6.7, for 'Aloe' image pair, the RCMP reaches the smooth peak when the half-window is between 4 and 8 (for those textureless image pairs, the half-window should be bigger to have their peak of RCMP). In fact, whatever the image type, the matching performance increases with aggregation window half-width in a certain range. Though on CPU and on GPU, all the single-precision floating-point computations follow the same accuracy standards. The accuracy-loss is more important on GPU owning to the accuracy problem of floating-point, that is why the results by CPU and that by GPU may have some deviations, especially when some cumulations in the computation exists. In our experiments, we use the floating point in the computation of matching cost with SSD and in the aggregation based on fixed support window, the deviation occurs at these computations, which can explain the tolerances between the RCMP by CPU and that by GPU shown in Fig. 6.7(b, d, f). What's more, the RCMP reaches the smooth peak when the half-window is between 4 and 8, while in some textureless image pairs, the half-window should be bigger to have their peak of RCMP. This implies that the size of the cost aggregation window has significant influence on the matching quality. Small windows do not contain enough information to allow a correct matching or for a unique minimum in the matching cost. At the opposite, too large aggregation windows may cover image regions containing pixels with different disparities, which violates the assumption of constant disparity inside the aggregation window. The size of the cost aggregation window should be well adjusted to make sure that only those useful pixels are covered, and it should be big enough to have sufficient information for a good stereo-matching result. Meanwhile, since the textures on the image vary their shapes, so, the aggregation window should be able to use the information of the textures for a preciser matching. An adaptive window could be more suitable for a better matching than the fixed window used in this implementation and the shifting window.

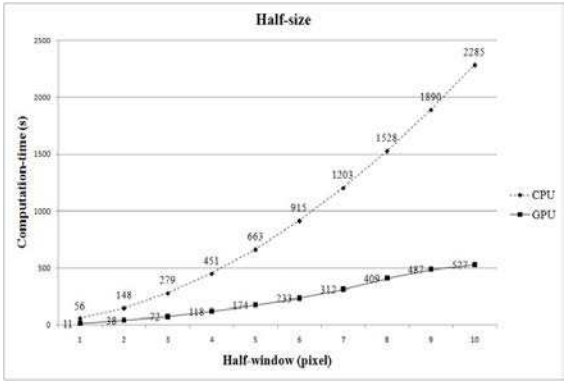
In addition, this method is also compared with the classic fixed windows matching methods, treating full color images and gray-level images, respectively, we used in former experiments. The results are shown in Table 6.1. The percentage of bad pixel (PBP) is the percentage of bad matched pixels in all the matched pixels and can be obtained by $PBP = 1 - RCMP$. This method performs worse on all these four pairs both in matching quality and in computation time. That is because this method requires too many refers to the logical operations in the programming and too many branches in data treatments, which are the real weaknesses of GPU architecture. These branches and logical operations lead to great load on to the GPU system when loading data from the memory space, and waste GPU's CUDA cores which are powerful in arithmetical operation, by making them do logical works.



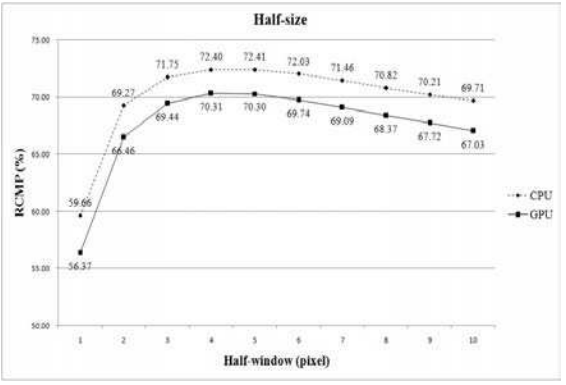
(a) Compute-time of GPU and CPU on full size datasets



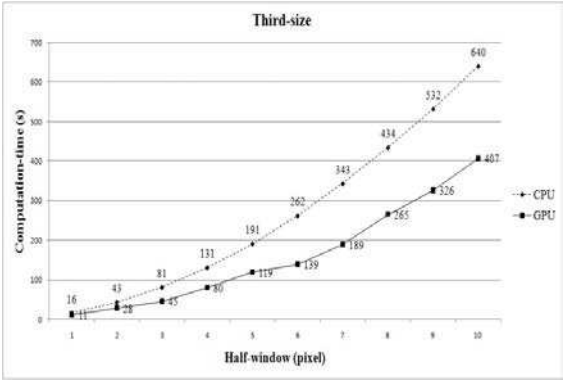
(b) RCMP of GPU and CPU on full size datasets



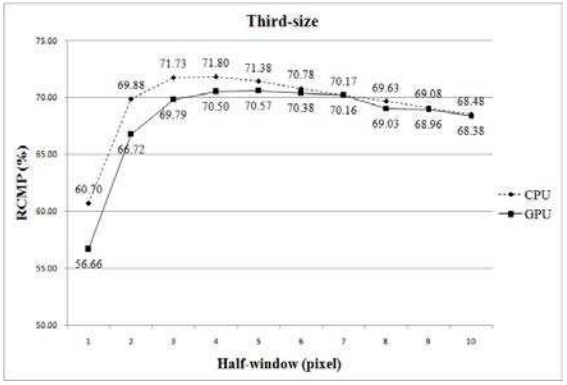
(c) Compute-time of GPU and CPU on half size datasets



(d) RCMP of GPU and CPU on half size datasets



(e) Compute-time of GPU and CPU on small size datasets



(f) RCMP of GPU and CPU on small size datasets

Figure 6.7: Computation-time and rate of correctly matched pixels (RCMP) obtained with the adapted SSD computed on the full size 'Aloe' stereo image pair for δ set to 1.

Table 6.1: COMPARATIVE EVALUATION ON THE FOUR MIDDLEBURY DATASETS OF THREE DIFFERENT SIZES BY DIFFERENT GPU PROGRAMMING VERSION. MEASURED BY 'COMPUTATION-TIME (CT)' AND 'PERCENTAGE OF BAD PIXEL (PBP)'

		Partial_Demosaiced		Full_Color		Gray_level	
Image	Size	PBP	CT (s)	PBP	CT (s)	PBP	CT (s)
Rocks1	Small size	28.76	0.214	25.81	0.191	24.58	0.146
	Half size	32.24	0.679	27.87	0.715	25.82	0.913
	Full size	37.06	5.811	31.20	5.717	28.93	8.235
Aloe	Small size	27.08	0.214	24.96	0.192	23.96	0.115
	Half size	29.33	0.698	25.78	0.697	24.94	0.68
	Full size	30.26	5.955	28.98	5.875	26.04	5.595
Cones	Small size	39.18	0.234	29.77	0.229	28.76	0.276
	Half size	46.98	2.123	37.36	1.548	36.91	3.024
	Full size	52.14	20.569	44.85	11.502	45.16	19.881
Teddy	Small size	45.27	0.23	32.23	0.23	29.86	0.284
	Half size	53.69	2.12	38.38	1.55	37.77	3.246
	Full size	57.36	27.09	46.58	11.51	47.37	24.922
Avg.		39.95	5.495	32.82	3.330	31.68	5.610

In the CFA demosaicing stereo-matching, the matching cost computation is done with the GPU parallel architecture, which is very powerful in massive mathematical computation. The used matching cost SSD, which needs the square computation, requires more operations in comparing with some other matching costs, such as the AD and SAD, which use only the basic arithmetic. The employment of a simpler matching cost measure could be helpful to accelerate the computation of matching cost.

During the stereo-matching, each thread is assigned to execute the whole processing including the matching cost computation, cost aggregation and WTA operation. The burden on each thread is significant. For the memory usage, the CFA demosaicing stereo-matching processing stores all the data on the global memory space. Each thread is in charge of one referred pixel in the left image, that means that for every candidate, the thread should access the global memory for intensity values needed in cost computation. Since the global memory access has a high latency, these accesses take quite much time, and furthermore, make the stereo-matching processing slower.

6.3/ REAL-TIME STEREO-MATCHING PROBLEM

6.3.1/ ACCELERATION MECHANISMS AND MEMORY MANAGEMENT

In last section, we have presented the color demosaicing stereo-matching method based on WTA optimization method. The experiment results show that this method, based on fixed cost aggregation window, does not match real-time requirement even though the employment of GPU does accelerate the stereo-matching processing. This problem leads us to look for a new color stereo-matching method for better matching quality and a faster

cost aggregation strategy to better profit from GPU's parallel architecture.

The new proposed method to reach real-time computation consists of a better memory management, and comports three important steps: new cost measure that includes support region information, called SAD-ALD cost measure, cost aggregation in adaptive window in cross-based support region, and a refinement step. These three steps are organized to be implemented in the GPU's parallel architecture.

For the purpose of improving the execution speed, we decided to partition the stereo-matching work into two parts: first part is the cost computation and the second part is the cost aggregation with WTA operation. Each of these two parts is assigned to one CUDA grid. So, the threads in the first CUDA grid will only take the work of computing the matching cost for each pixel with a set of candidates, and the second CUDA grid will sum up the matching cost of one pixel and choose the best candidate as the estimated disparity. A logical 3D memory space will be employed to store matching costs obtained by the first grid. In the second part, each thread sums up its assigned pixel's cost values. Data reuse with shared memory is considered in this step to reduce the accesses into the global memory space.

The method is a correlation-based technique, which falls into the class of local dense stereo-matching approaches, and it includes the following key characters:

- A better memory management by using a 3D cost volume of size $W \times H \times R$, where R is the maximum disparity range, and W, H the image size.

- SAD-ALD cost measure combining the adapted sum of absolute differences (SAD) measure and the arm-length-differences (ALD), where the ALD is the difference of the adaptive window vertical lengths.

- Adaptive cross-based region for cost aggregation. Proposed by Zhang *et al.* [ZLL09], this support region is based on a cross skeleton.

- A simple refinement process with support region voting that helps repair wrong matched pixels.

- Efficient system implementation based on CUDA programming.

The parallelism brought by GPU architecture and CUDA implementation provides significant acceleration in running time. This method is tested on six pairs of images from Middlebury database, and for each pair of images it generates acceptable matching results in less than 100 milliseconds. The method is compared with three different versions of the previous CFA-implementation, and one CPU-based Dynamic Programming method, on increasing size images. The SAD-ALD cost measure was able to provide more accurate matching results than the fixed window aggregation method.

Section 6.3.2 presents the stereo-matching steps of the proposed method, including the matching cost computation, cost aggregation strategy and the refinement step. Section 6.3.3 details the experiments carried out on a set of increasing size images from Middlebury database and against other GPU and CPU standard methods.

6.3.2/ ADAPTIVE STEREO-MATCHING SOLUTION

6.3.2.1/ MATCHING COST

The method is a local dense stereo-matching method. It respects the very assumption that the color intensity of neighbors of a left pixel should be close to those of the same neighbors of its homologous right pixel in the right image. So the matching costs are defined between the left pixel and the candidate right pixels in the corresponding line (epipolar line) in the right image. The matching cost between the left pixel and each candidate in the right image is computed. By the aggregation of matching cost and the winner-takes-all (WTA) method, the final disparity estimation is realized by selecting the candidate pixel with the lowest matching cost. In this application, we still use the WTA optimization method to choose the best candidate pixel in the right image for every referred pixel in the left image.

The matching cost used here is Sum of Absolute Differences (SAD) of the three color components at each pixel, adapted as Equation (6.6) from Equation 3.6.

$$SAD(x_l, y, s) = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \times \begin{bmatrix} |I_{lR}(x_l, y) - I_{rR}(x_l - s, y)| \\ |I_{lG}(x_l, y) - I_{rG}(x_l - s, y)| \\ |I_{lB}(x_l, y) - I_{rB}(x_l - s, y)| \end{bmatrix} \tag{6.6}$$

Where $|\cdot|$ is the absolute value, while s is the spatial shift along the horizontal epipolar line (or we call it the disparity of the two pixels) and $|I_{li}(x_l, y) - I_{ri}(x_l - s, y)|_{(i=R,G,B)}$ is the absolute difference (AD) of three color components in the two chosen pixels. For the coefficient matrix of the three color components, we choose the $\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$ according to the Bayer Filter Mosaic, which uses twice as many green elements as red or blue to mimic the physiology of the human eye.

Since these pixels of horizontal lines with the same parity in the left and right color images are characterized by the same three color components, we can reasonably assume that the color points of two homologous pixels are similar. Because the matching cost compares the color points of left pixels and right pixels located on the same horizontal lines, it reaches an extremum when the shift s is equal to the disparity.

An assumption for matching cost aggregation is that a pixel and its homologous pixel should have the similar support region in vertical direction. This implies that the information about support region can be used to enhance the matching results in most regions of the image pairs.

Given an image, an upright cross support region is constructed for every pixel. The support region of a given pixel, such as p in Fig. 6.8, is modeled by merging the horizontal arms of the pixels (the pixel q in Fig. 6.8) lying on the vertical arms of pixel p. Generally, every pixel has four arms and the length of the arms is set by an endpoint pixel p' in the same direction that does not obey both the two following rules:

1) $D_c(p, p') < \sigma_{dc}$ and $D_c(p', p'') < \sigma_{dc}$, where $D_c(p, p') = max|I_i(p) - I_i(p')|_{(i=R,G,B)}$ is the color difference between the pixel p and the pixel p', and p'' is the predecessor of p' lying between p and p' while σ_{dc} is an empirical preset threshold value.

2) $D_s(p, p') < \sigma_{ds}$, where $D_s(p, p') = |p - p'|$ is the spatial distance, which is equivalent

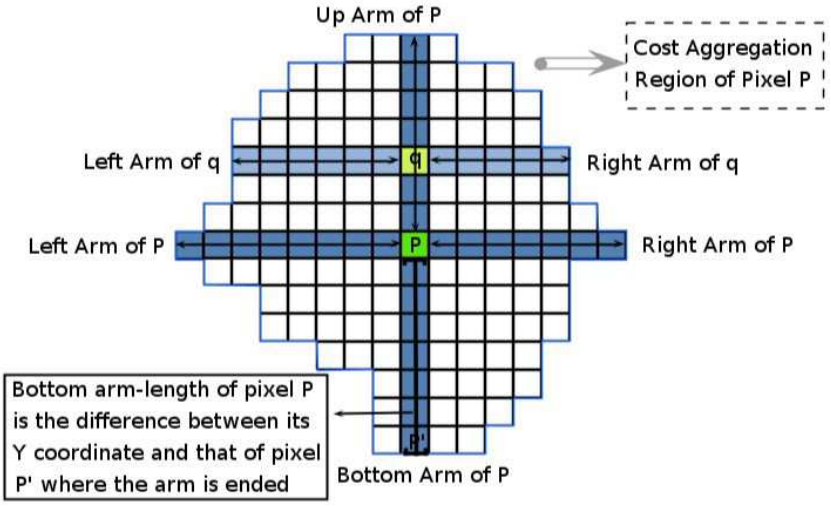


Figure 6.8: Cross construction of central pixel's support region.

to absolute difference of their coordinates in the same direction ($|x_p - x_{p'}|$ or $|y_p - y_{p'}|$), while the σ_{ds} is an empirical preset maximum length measured in pixels.

These two rules provide constraints in the four arm directions both on color similarity and arm length with parameter σ_{dc} and σ_{ds}. After the cross construction step, the support region for pixel p is modeled by merging the horizontal arms of all the pixels lying on $p's$ vertical arms, as done for q for example in Fig. 6.8.

As it is illustrated in Fig. 6.8, for a given pixel p with a support region, we define the arm-length (AL) as equivalent to absolute difference of the coordinates in the same direction $AL = |y_p - y_{p'}|$ in vertical direction and $AL = |x_p - x_{p'}|$ in horizontal direction. $ALD = |AL_l(x_l, y) - AL_r(x_l - d, y)|$ is the difference of arm-length (ALD) between homologous pixels. Since there exist some phenomenons, such as the half-occlusion, where the homologous pixels' horizontal arms could be different, however, their vertical arms should be always similar. So, we update the matching cost as Equation (6.7) with the up arm-length and the bottom arm-length.

$$MatchingCost(x_l, y, s) = SAD(x_l, y, s) + K \times (ALD_{up} + ALD_{bottom}) \tag{6.7}$$

where the parameter K is an empirical preset value, the ALD_{up} and ALD_{bottom} are the ALD for the up arm and the bottom arm, respectively.

6.3.2.2/ UPDATED COST AGGREGATION

In this step, each pixel's matching cost over its support region is aggregated from the initial cost volume in order to pick out the best candidate pixel.

Zhang *et al.* [ZLL09] have proposed a cross-based matching cost aggregation method. We adapt this method to GPU's parallel computation architecture.

The cross-based aggregation is carried out by a two-step-process. The first step is to construct a support region, as is explained in last section. In the second step, referring

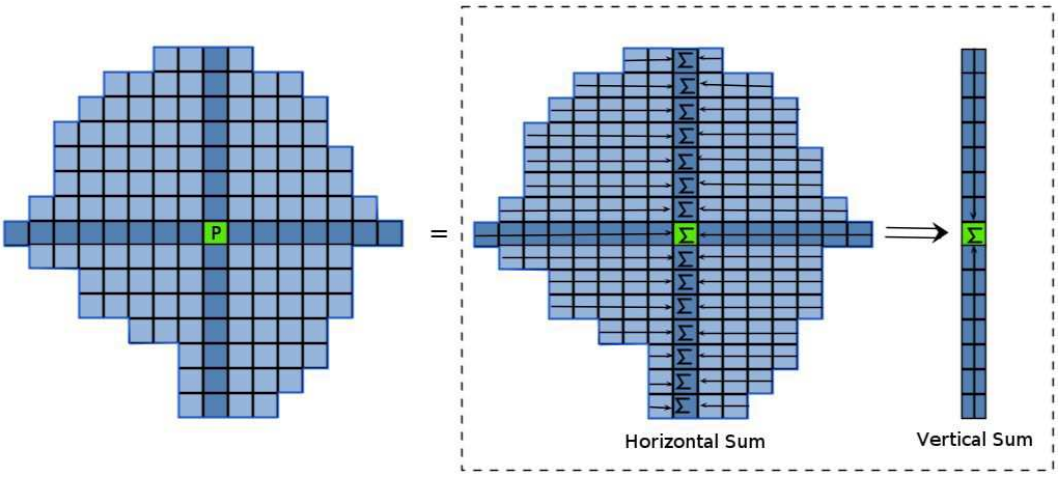


Figure 6.9: Cross-based cost aggregation.

to Fig. 6.9, the aggregated costs over all pixels are computed by firstly summing up the matching costs horizontally and secondly summing up these horizontal sum results vertically to get the final costs.

However, in some too textured regions, the color and the shape both repeat, which leads to degradation in matching results. Meanwhile, for some region with very small texture shapes, the matching quality also decreases. We find that the reason for these degradations lies in the shape of aggregation support region. As shown in Fig. 6.8, we take the pixel at the end of pixel $p's$ right arm as an example. For this pixel, its up arm and bottom arm could be very short (less than 2 pixels). So, in the aggregation of its matching cost, there will be not enough information to achieve a unique minimum in the Winner-Takes-All processing, which leads to marching errors at this pixel. As a solution to this problem, we artificially enlarge the arms of a pixel by two pixels, if its support region is too small to make sure the matching cost aggregation processing can have enough information for stereo-matching. This operation provides a slight improvement in matching results in paying no computation time cost.

6.3.2.3/ SIMPLE REFINEMENT

After the previous step, the disparity maps of both the left image and the right image contain some outliers in certain regions that should be corrected by further operations. A simple refinement is carried out after detecting these outliers.

The outliers in the estimated disparity maps are detected with left-right consistency check as introduced in Section 3.3.6. These detected outliers are these errors that should be corrected. The most current accurate stereo-matching algorithms use segmented regions for outlier handing [Hir08, YWY+09], which are not suitable for GPU architecture. Here, what we use is a simple voting refinement in reusing the support region information. We still take the pixel p in the left image as an example, all the reliable disparities lying in its cross-based support region are sorted by their disparity values. The disparity value which repeats the most (has the most votes) is denoted as $\hat{d'}_{Lp}$. Its repeating frequency is denoted as $F_p(\hat{d'}_{Lp})$. The number of reliable pixels are denoted as S_p^r and the total

number of pixels in its support region are denoted as S_p. The disparity value of outlier pixel p is then replaced with $\hat{d'}_{Lp}$ if these inequations in Equation 6.8 and 6.9 hold true, where, σ_F and σ_S are empirical preset threshold values. If not, the $p's$ disparity will be updated with nearest reliable disparity [SS02] in its support region.

$$\frac{F_p(\hat{d'}_{Lp})}{S_p^r} > \sigma_F \tag{6.8}$$

$$\frac{S_p^r}{S_p} > \sigma_S \tag{6.9}$$

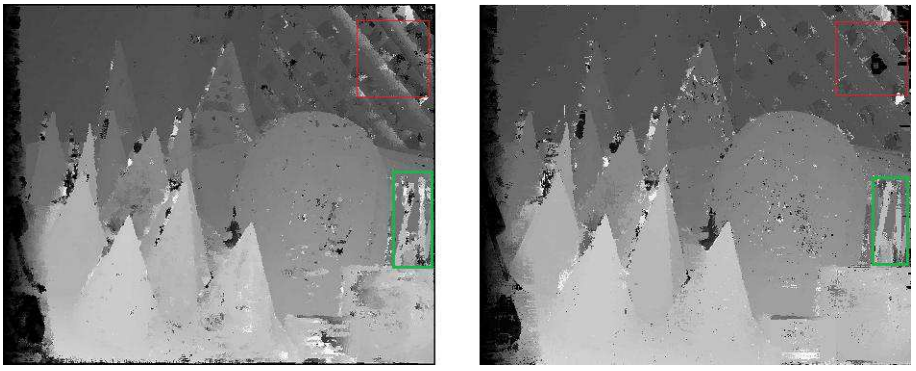


Figure 6.10: Improvement result from the different enhancements in stereovision processing.

We found that the different enhancements in cost computation, adaptive window aggregation and refinement step can evidently reduce the errors in most regions, as shown in Fig. 6.10. As shown in the figure, the ambiguities in the regions covered by red and green frames are reduced.

6.3.3/ EXPERIMENT

To access the efficiency of our stereo model, we compare its performance with those reached by some other methods, such as the CFA demosaicing color stereo-matching method, the standard method for color images and the dynamic-programming method. The experiments are carried out with the image pairs from Middlebury's database [HS06]. Each dataset of this database is made up of a pair of stereo images and the ground-truth disparity map.

Among the 24 datasets provided in this database, we have selected six pairs ("Rocks1", "Aloe", "Tsukuba", "Cones", "Teddy" and "Venus") for the experiments in this section. Fig. A.4 in Appendix A shows the contents of these six image pairs: the left image, the right image and the ground-truth disparity map.

6.3.3.1/ CUDA IMPLEMENTATION

Given an image with size of $W \times H$, we briefly lay out the CUDA settings and the parameter values in our algorithm.

In our experiments, we use two-dimension *blocks* with size of 16×16. Every thread takes care of one pixel at the three steps: matching cost computation, cost aggregation and refinement. So, for the $W \times H$ image, there are three CUDA *grids* containing $W \times H$ threads distributed to the three matching steps. The size of *grid* is obtained by $\frac{W+16-1}{16} \times \frac{H+16-1}{16}$.

In the cost computation step, a *grid* is created with $W \times H$ threads. Every thread takes care of one pixel for the matching cost value computation at a set of given disparities. These cost values are kept in the memory space for the following steps. In the cost aggregation step, the kernel employs one thread for one pixel to take care of its matching cost aggregation and the Winner-Takes-All processing aiming at a winner pixel from a set of candidate pixels. Here, more data will be loaded into the on-chip memory space for fast access, and the use of shared memory is considered.

For the simple refinement, the platform does the executions concurrently on the estimated disparity images. A third *grid* with size of $W \times H$ is employed to make sure that each pixel has one thread for its refinement processing.

The experiment parameters are given in Table 6.2 which are kept constant in all the experiments.

Table 6.2: EXPERIMENT PARAMETERS.

K	σ_{dc}	σ_{ds}	σ_F	σ_S
1.12	12	10	0.4	0.55

6.3.3.2/ EXPERIMENT RESULTS

We tested our method on the standard image pairs from the Middlebury datasets. The four pairs of images with three different sizes (measured in pixels) are shown in Table 6.3

Table 6.3: EXPERIMENT INPUT IMAGE SIZES, MEASURED IN PIXELS.

	Small size	Half size	Full size
Rocks1	425×370	638×555	1276×1110
Aloe	427×370	641×555	1282×1110
Cones	450×375	900×750	1800×1500
Teddy	450×375	900×750	1800×1500

We first compare our method with three other methods that we implement also on GPU. These methods are the partial demosaicing matching method originally executed on GPU in Section 6.2 and the classic fixed windows matching methods treating full color images and gray-level images, respectively. The results are shown in Table 6.4. The PBP column reports the percentage of bad pixels (PBP), whereas the CT column reports the computation time in seconds. We can verify that our adaptive method competes with the other ones on all these four pairs both in matching quality and in computation time, as it is shown in Fig. 6.11 more intuitively. It is worth noting that near-real time computation is achieved for the small size images, since computing time may reduce to about less than 100 ms. Not like most other methods in the Middlebury evaluation based on GPU,

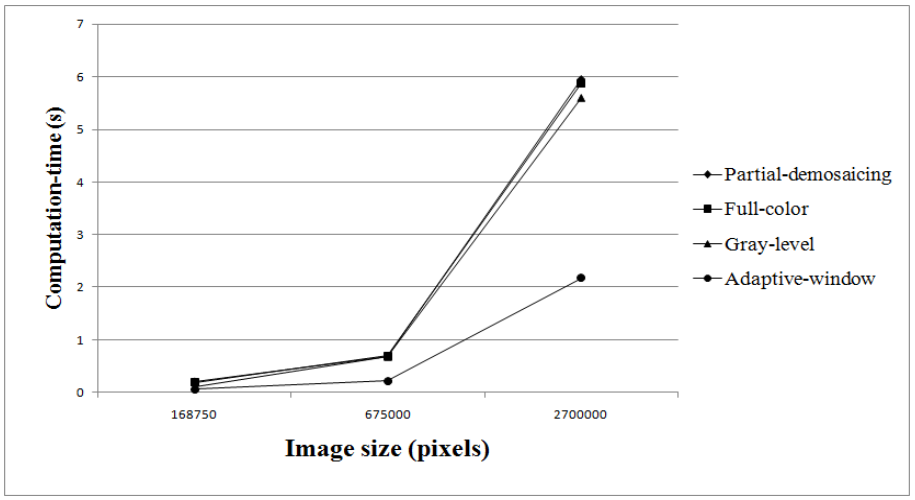
our method puts most of the data on the global memory space and carefully treats the coalescence of memory access with proper programming structures and adapted usage of cached memory space of GPU such as shared memory and texture memory, which makes our method very extensible and scalable for large image pairs.

Table 6.4: COMPARATIVE EVALUATION ON THE FOUR MIDDLEBURY DATASETS OF THREE DIFFERENT SIZES. MEASURED BY 'COMPUTATION TIME (CT) ' AND 'PERCENTAGE OF BAD PIXELS (PBP)' .

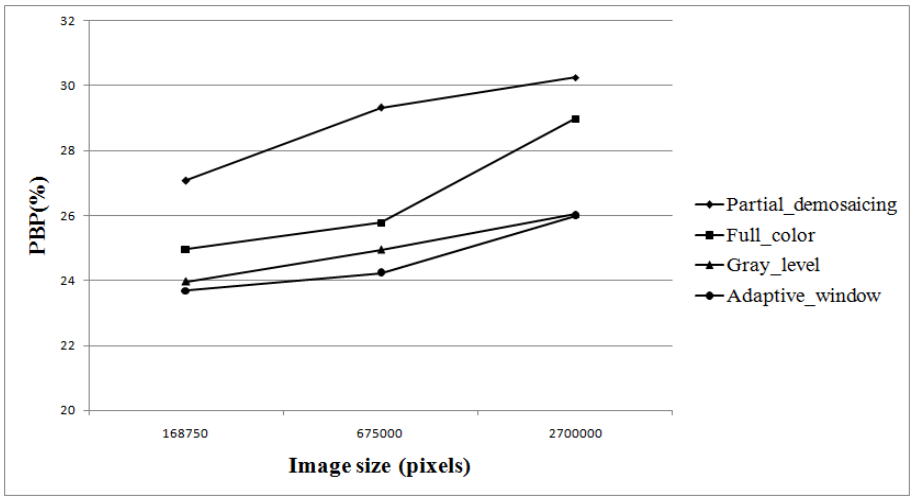
		Partial_Demosaiced		Full_Color		Gray_level		GPU_Ada	
Image	Size	PBP	CT (s)	PBP	CT (s)	PBP	CT (s)	PBP	CT (s)
Rocks1	Small size	28.76	0.214	25.81	0.191	24.58	0.146	24.28	0.068
	Half size	32.24	0.679	27.87	0.715	25.82	0.913	24.91	0.239
	Full size	37.06	5.811	31.20	5.717	28.93	8.235	28.40	2.275
Aloe	Small size	27.08	0.214	24.96	0.192	23.96	0.115	23.69	0.063
	Half size	29.33	0.698	25.78	0.697	24.94	0.68	24.22	0.217
	Full size	30.26	5.955	28.98	5.875	26.04	5.595	25.99	2.180
Cones	Small size	39.18	0.234	29.77	0.229	28.76	0.276	18.90	0.079
	Half size	46.98	2.123	37.36	1.548	36.91	3.024	35.26	0.812
	Full size	52.14	20.569	44.85	11.502	45.16	19.881	44.18	6.159
Teddy	Small size	45.27	0.23	32.23	0.23	29.86	0.284	23.67	0.112
	Half size	53.69	2.12	38.38	1.55	37.77	3.246	36.09	0.999
	Full size	57.36	27.09	46.58	11.51	47.37	24.922	45.98	6.783
Avg.		39.95	5.495	32.82	3.330	31.68	5.610	28.84	1.666

We also compared our method to a standard dynamic-programming (DP) matching method on CPU. The results are presented in Table 6.5, and in Fig. 6.12 and Fig. 6.13 more intuitively. These two methods can achieve similar matching quality, but our method outperforms the dynamic-programming method in computation time with about five to ten times acceleration. On the four small size image pairs, the dynamic-programming method can finish its work in less than half a second, however, our system has work done within 100 milliseconds.

Some disparity results are presented in Fig. 6.14. These results concern the four images allowed in the Middlebury database for general comparison and ranking. These images are the small size images 'Tsukuba', 'Venus', 'Teddy' and 'Cones'. The ranking evaluations obtained from Middlebury database are shown in Fig. 6.15. Our method gives back its best results for the 'Venus' image pair. Generally speaking, the matching quality of the method is not very competitive in comparing with some other very sophisticated methods on CPU, especially for the 'Tsukuba' image pair. In some regions of this image pair, near the shoulder or the lamp for example, the color is too dark and the color components' values are far out of the ordinary, this introduces noise to the matching method. Different from the methods on CPU, that take at least half a second to do the stereo-matching, our method requires only 0.017 seconds for 'Tsukuba' pair, 0.053 seconds for 'Venus' pair, 0.079 seconds for 'Cones' pair, and 0.112 seconds for 'Teddy' image pair. Our method on GPU brings significant speedup even comparing to dynamic-programming method on CPU. While in running time, the step of cost aggregation occupies the biggest proportion.

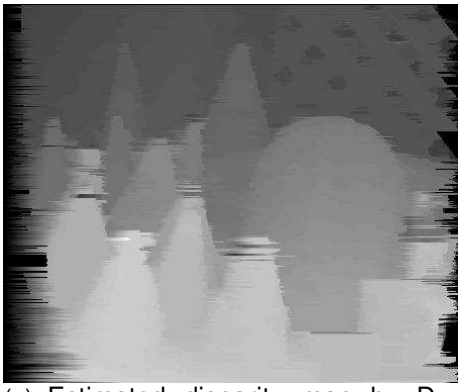


(a) Comparison on computation-time (s).

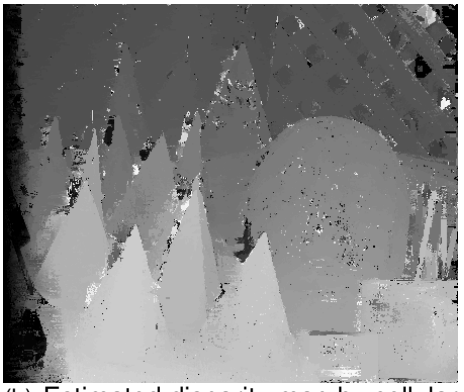


(b) PBP(%) comparison results.

Figure 6.11: Comparison of the methods on the ‘Aloe’ image pair.

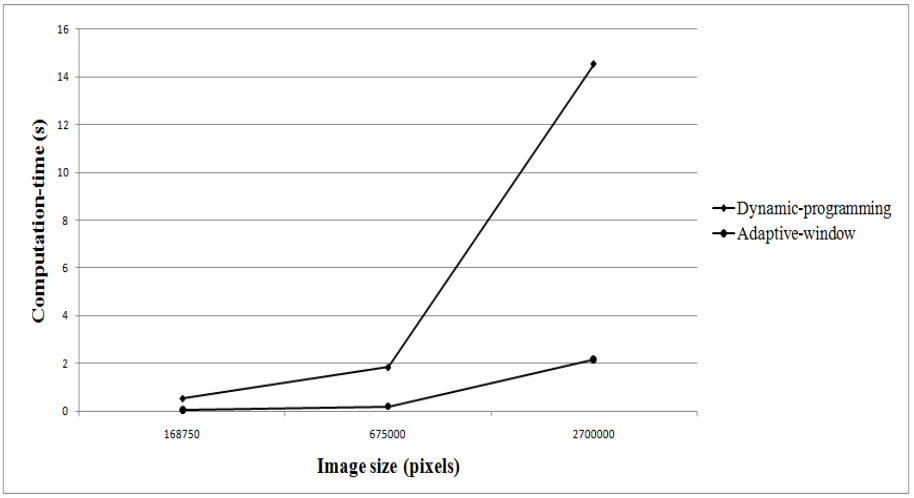


(a) Estimated disparity map by Dynamic Programming.

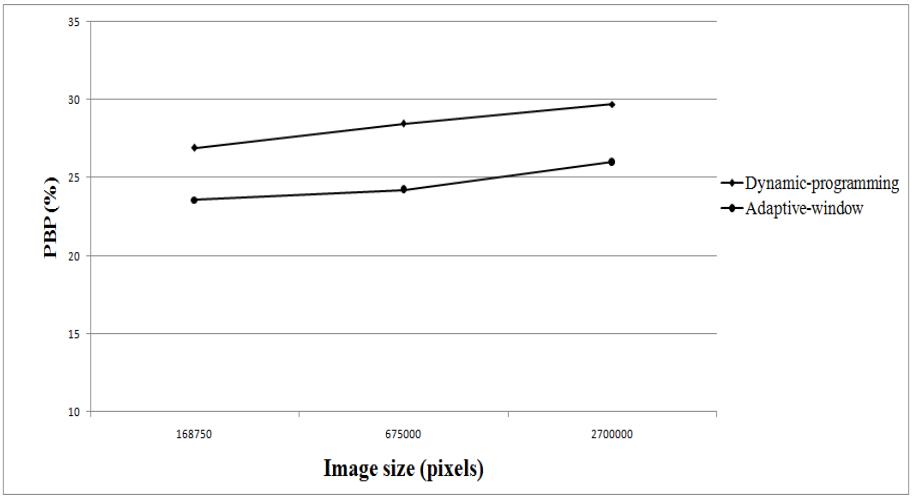


(b) Estimated disparity map by cellular method.

Figure 6.12: Visualization of the density maps on the ‘Cones’ image pair.



(a) Comparison on computation-time (s).



(b) PBP(%) comparison results.

Figure 6.13: Comparison to Dynamic Programming on 'Aloe' image pair.

Table 6.5: COMPARATIVE EVALUATION WITH DP ON THE FOUR MIDDLEBURY DATASETS OF THREE DIFFERENT SIZES. MEASURED BY 'COMPUTATION TIME (CT) ' AND 'PERCENTAGE OF BAD PIXELS (PBP)' .

		CPU_DP[1]		GPU_Ada[2]	
Image	Size	PBP	CT (s)	PBP	CT (s)
Rocks1	Small size	25.49	0.509	24.28	0.068
	Half size	27.71	1.811	24.91	0.239
	Full size	27.85	14.194	28.40	2.275
Aloe	Small size	26.90	0.544	23.69	0.063
	Half size	28.46	1.842	24.22	0.217
	Full size	29.72	14.534	25.99	2.180
Cones	Small size	21.57	0.583	18.90	0.079
	Half size	28.89	4.903	35.26	0.812
	Full size	35.32	37.736	44.18	6.159
Teddy	Small size	20.80	0.590	23.67	0.112
	Half size	27.95	4.881	36.09	0.999
	Full size	33.51	37.704	45.98	6.783
Avg.		27.85	9.986	28.84	1.666

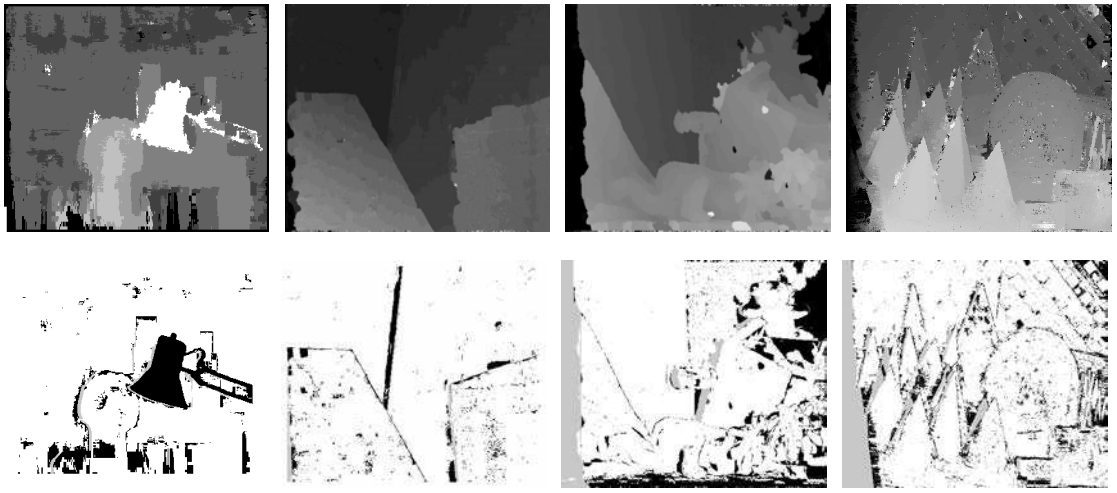


Figure 6.14: Matching results for the four basic Middleburry image pairs 'Tsukuba', 'Venus', 'Teddy' and 'Cones'. Estimated disparity map in first row and disparity matching error maps in second row with threshold 1, where the errors in unoccluded and occluded regions are marked in black and gray separately.

Algorithm	Avg. Rank	Tsukuba ground truth nonocc	Tsukuba all	Tsukuba disc	Venus ground truth nonocc	Venus all	Venus disc	Teddy ground truth nonocc	Teddy all	Teddy disc	Cones ground truth nonocc	Cones all	Cones disc	Average Percent Bad Pixels
FW-DLR [129]	141.4	4.87 139	5.89 133	22.9 148	2.50 130	3.22 128	18.3 132	18.2 149	18.7 130	37.2 150	24.2 153	27.9 152	42.1 153	18.8
SO [1c]	142.9	5.08 141	7.22 145	12.2 113	9.44 150	10.9 150	21.9 139	19.9 151	28.2 153	26.3 135	13.0 147	22.8 150	22.3 141	16.6
MI-nonpara [85]	145.2	5.59 145	7.54 147	18.8 139	7.50 146	8.99 146	35.0 150	17.4 146	25.7 149	36.9 149	10.2 140	19.9 143	22.6 143	18.0
OUR METHOD	145.8	14.4 153	15.5 153	38.2 153	4.60 141	5.95 142	28.3 146	16.3 142	23.9 142	30.8 143	12.1 144	21.7 147	22.9 144	19.6
PhaseDiff [23]	146.2	4.89 140	7.11 143	16.3 131	8.34 149	9.76 149	26.0 144	20.0 152	28.0 152	29.0 141	19.8 152	28.5 153	27.5 149	18.8
STICA [16]	146.4	7.70 151	9.63 152	27.8 150	8.19 148	9.58 147	40.3 153	15.8 140	23.2 141	37.7 151	9.80 137	17.8 136	28.7 151	19.7
Rank+ASW [84]	146.5	6.51 149	8.43 149	19.7 141	10.5 152	12.0 152	32.7 148	15.7 139	24.1 144	32.8 145	14.1 148	23.1 151	21.7 140	18.4
LCDM+AdaptWgt [75]	146.8	5.98 148	7.84 148	22.2 146	14.5 153	15.4 153	35.9 151	20.8 153	27.3 151	38.3 152	8.90 134	17.2 135	20.0 137	19.5
Infection [10]	148.1	7.95 152	9.54 151	28.9 152	4.41 140	5.53 140	31.7 147	17.7 147	25.1 148	44.4 153	14.3 149	21.3 146	38.0 152	20.7

Figure 6.15: The rankings of *our method* in the Middlebury database with the error percentages in different regions.

6.4/ CONCLUSION

In this chapter, we have presented two stereovision methods that fit into a general GPU cellular parallel model: the demosaicing stereo-matching method on CFA images and the real-time stereo-matching method based on adaptive cost aggregation.

In the first implementation, we have presented a partial demosaicing scheme specially designed for stereo-matching of CFA images. By analyzing the CFA image directly, this method can handle the stereo-matching in case of single-CCD cameras usage. This method has three main technique characters: the matching cost for CFA image, the estimated second color component based on Hamilton's estimate method, and cost aggregation window. The results show that this method is well suitable for GPU's parallel architecture but has some limits since it does not match real-time computation, and solution quality should be improved. Meanwhile, the cost aggregation is carried on fixed windows, which leads to errors in textured regions. So, a more efficient and accurate aggregation strategy should be devised for computation acceleration and quality improvement.

In the second implementation, we present a stereo-matching method suitable to GPU's parallel architecture with good performance, when looking at the trade-off between accuracy and computation time. The method is formed of three steps: SAD-ALD cost measure computed in 3D cost volume, cost aggregation in adaptive window in cross-based support regions and a refinement step to reduce the matching errors in the disparity results. Every step is completely distributed between the many threads so that this method clearly fits to the general cellular parallel model. Experiment results show the accuracy and the efficiency of the method: it can handle the four pairs of images from Middlebury dataset within roughly 100 milliseconds, with acceptable matching quality both in non-occluded regions and depth discontinuities. Furthermore, the approach scales well as the image sizes increase, even if the volume cost that depends on the disparity range increases the memory consumption.

Although the running time is short, the implementation in real-time is still a great challenge. As the cost aggregation step takes the biggest proportion of running time, looking for a more efficient way to further accelerate cost aggregation and finding out a set of robust experiment parameters to improve matching quality can be interesting topics for fu-

ture studies. The performance of these stereo-matching methods shows that the parallel model based on GPU's multi-cores can well adapt the classic stereo-matching methods, and can achieve high acceleration in treating such stereo-matching problems. Since the images can be seen as representing Euclidean spaces, we can now see in next chapter how structured meshing generation or transportation routing problem in plane fit into the cellular parallel model.

7

GPU IMPLEMENTATION OF CELLULAR MESHING ALGORITHMS

7.1/ INTRODUCTION

In this chapter, we give the details about the two GPU applications of the cellular parallel SOM algorithm. The two applications are the balanced structured mesh problem applied on disparity map, and the well-known Euclidean traveling salesman optimization problem. For both applications, the solution we propose is based on the Kohonen's self-organizing map learning algorithm because of its ability to generate a topological grid that reflects a distribution map, and its ability to be a natural massive parallel algorithm. Here, we detail implementation on GPU with CUDA code interface.

We present experiments for the structured mesh generation. We detail GPU implementation and experimental results. Here, the disparity map stands for a density distribution that reflects the proximity of objects to the camera in 3D space. The goal is to generate a compressed structured hexagonal mesh where the nearest objects are provided with more details than objects which are far from the camera. The grid is then used to reconstruct and represent the 3D surface of the scene. We also present experiments for the Euclidean TSP. The approach is applied to instances of well-known databases. That are, Euclidean TSPLIB problems and National TSPs with up to 33708 cities. Tests are carried out on both GPU and CPU, and these two types of implementation are compared and discussed.

In complement, this section includes the following topics:

- Discussion about implementation with CUDA code, memory management, warp divergence.
- Evaluation on elaborate experiments and comparison with regard to both solution quality and running time.
- Systematic comparison to serial implementation according to the increase of input size.

The chapter contains three main sections. We first present the common part of the GPU/CUDA algorithm for both applications. Then, we detail experiments for the balanced structured meshing application, and for the traveling salesman problem application, respectively.

7.2/ GPU IMPLEMENTATION OF PARALLEL SOM

7.2.1/ PLATFORM BACKGROUND AND MEMORY MANAGEMENT

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 CPU 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.

We use the Compute Unified Device Architecture (CUDA) programming interface for GPU to implement our SOM parallel model for both balanced structured meshing and traveling salesman problem. In the CUDA programming model, the GPU works as a SIMT co-processor of a conventional CPU. 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. Hence, we apply CUDA threads as the parallel processing units in our model.

All CUDA threads are organized into a two level concepts: CUDA grid and CUDA block. A kernel has one grid which contains multiple blocks. Every block is formed of multiple threads. The dimension of grid and block can be one-dimension, two-dimension or three-dimension. Each thread has a *threadId* and a *blockId*, which are built-in variables defined by the CUDA runtime to help user locate the thread's position in its block as well as its block's position in the grid.

In our applications, we use two-dimensional bocks that are adjusted in size for best performance, knowing that the total number of threads (cell matrix size) will always be dependent on the problem size only. In order to improve coalescing memory accesses, we systematically organized data structure tables in such way that two consecutive threads necessarily address two consecutive memory locations in a table. In other words, the indexes to address memory tables are identical to the indexes of the threads themselves. In that way, the next thread systematically refers to the next data location in a same table. Also, we will see that thread divergence often occurs, since branching instructions strongly depend on data distribution. We will study divergence effects for the TSP application specifically.

7.2.2/ CUDA PROGRAM FLOW

The CUDA program flow of GPU implementation is presented in Algorithm 5. Lines 2, 4, 7, 8, 11, and 13 are implemented with CUDA kernel functions that will be executed by GPU threads in parallel. The kernel function in Line 2 is used for calculating each cell's density value, i.e. the number of element points or sum of pixel intensities in each cell. After all the cells' density values are obtained, the maximum one is found. This last work in Line 3 is done on CPU since it is done only one time and does not directly concern the main behavior. Note that computing a maximum value is a trivial job even when done on GPU. Then, the cells' activation probabilities are computed according to the activation

Algorithm 5 CUDA program flow.

```
1: Initialize data and perform CPU to GPU data transfers;
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:       Set seeds for random number generators;
8:       Generate random numbers;
9:    end if
10:   if ite% CELL_REFRESH_RATE == 0 then
11:      Refresh cells;
12:   end if
13:   Parallel SOM process;
14:   Modify SOM parameters (radius and intensity);
15: end for
16: Perform GPU to CPU data transfers and save results;
```

formula of Equation 5.6 by the kernel function of Line 4. In each iteration of the program, each cell needs two random numbers: one is used for cell activation and the other is used to extract input point in the activated cell. With respect to the large scale input instances with huge cellular matrix and numerous iterations, the random numbers generated via kernel functions shown in Line 7 and Line 8 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 of constant rate factor called $MEMORY\_REUSE\_SET\_RATE$. The random number generators we use in Line 7 and Line 8 are from Nvidia CURAND library [NVI12a].

Line 10 and Line 11 concern the cell refreshing. Each cell has data structures where to deposit information of the number and indexes, in the neuron grid, of the grid nodes it contains. This information may change during each iteration, since nodes continuously move on the plane and may change their cell locations. It appears by experiments, that it can be sufficient to make the refreshing of the cell contents based on a lower rate refresh rate coefficient, called $CELL\_REFRESH\_RATE$. The cell contents are refreshed via kernel function in Line 11. Note that neurons' locations are moved in the plane at each single iteration, whereas their indexes memorized in cells are only refreshed based on a lower rate. Then, the parallel SOM process takes place with kernel function of Line 13. After the parallel SOM process is done, the SOM parameters will be modified getting prepared to do the next iteration. Note that no data transfers occur between CPU and GPU, in both senses into the main loop, except the passing of kernel parameters, such as data pointers and SOM parameters, that are neighborhood radius and intensity for learning rule.

As shown in Algorithm 5, the four GPU kernel functions: Seed setup, Random number generation, Cell refresh, and Parallel SOM are launched many times during the program. We calculate each kernel's running time with CUDA event API [NVI12a]. A running time statistical result of the program handling $pcb442$ TSP instance with 100000 iterations is shown by Table 7.1. The data, which is mean value of 10 runs, shows that the most time consuming kernel clearly is parallel SOM. More thorough examination shows that cellular

Table 7.1: RUNNING TIME OF EACH KERNEL.

Kernel name	Running time (ms)
Seed setup	49.48
Random number generation	51.44
Cell refresh	1062.48
Parallel SOM	3631.23

spiral search is the most consuming part of the parallel SOM.

Overall, the host code (CPU side) of the program is mainly used for flow control and the entire GPU threads synchronization by sequentially calling separate kernel functions. For all the kernel functions, one thread handles one cell and the number of threads launched by each kernel is no less than the number of cells.

7.2.3/ PARALLEL SOM KERNEL

Algorithm 6 GPU parallel SOM Kernel flow.

```
1: Locate cell position associated to current thread;
2: Check if the cell is activated;
3: if the cell is activated then
4:     Select a point in the cell, randomly or by roulette wheel;
5:     Perform a spiral search within a certain range;
6:     Modify positions of the winner neuron and its neighbors;
7: end if
```

The parallel SOM kernel function is applied in Line 13 of main CUDA Algorithm 5 and is further illustrated by Algorithm 6. Firstly, it locates the cell's position by its *threadId* and *blockId*. Then, the thread checks if the cell is activated or not, by comparing the cell's activated probability to a random number with value between 0 and 1. If the cell is activated, the thread randomly selects an input data point covered by the cell by using a second random number with value between 0 and the cell's density value (number of points or sum of pixel intensities). If the input are pixels, a roulette wheel according pixel intensity is performed to select to input pixel point. If the input are cities, a simple uniform choice is done to get the city point. After that, the thread performs a spiral search within a certain range on the grid for finding the closest neuron to the selected point. The maximum number of cells that a thread has to search equals $(range \times 2 + 1)^2$. As shown in Fig. 7.1, if the spiral search range is set to 2, the thread will search in current cell, the eight cells around, and the sixteen cells around the first range. The number of cells a thread has to search equals $(range \times 2 + 1)^2$. After finding the winner neuron, the thread carries out learning process via modifying positions of the winner neuron and its neighbors. 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 one same neuron's location at the same time, which causes race conditions. In order to guarantee a coherent memory update, we use the CUDA atomic functions, which can perform a read-modify-write atomic operation without interference from any other threads.

2	2	2	2	2
2	1	1	1	2
2	1	0	1	2
2	1	1	1	2
2	2	2	2	2

Figure 7.1: Spiral search range.

7.3/ APPLICATION TO BALANCED STRUCTURED MESH PROBLEM

7.3.1/ CUDA IMPLEMENTATION SPECIFICITIES

Based on our experimental platform, given an image of size $W \times H$, we first determine the size of the cellular matrix that defines the total number of threads used. To each cell of the cellular matrix is assigned a single thread. Since we want a linear relationship between the thread number and the image size, we divide each dimension of the image by a constant factor that will be the same for all experiments. Based on such image decomposition between threads, we define the size of each CUDA block, and deduce the number of blocks necessary, hence the size of the CUDA grid. It appears in experiments, because of thread divergence within blocks and hence warps, that the best performance was reached with blocks having each a single thread. Hence, only parallelism from independent streaming multiprocessors appears to have some important benefit in this implementation. Threads within warps inside a same block are thus subject to warp divergence that may introduce serialization on the computation.

The CUDA code follows the CUDA program in Algorithm 5. It is composed of an initialization phase, followed by the iteration of the main loop of the algorithm. At initialization, two kernels compute the random numbers activation probabilities, that are assigned to each cell. Also, a kernel for random number generation is called at the beginning of the simulation. The data generated by these kernels are kept on the global memory space for following steps. Into the main loop iteration step, two kernels are applied to carry out the SOM processing. The first kernel is responsible to refresh the cellular matrix at a given lower rate. The second kernel handles the main process of SOM: the extraction of pixel, finding the closest topological grid node by spiral search, and learning step. Note that these two kernels are executed t_{max} times, and that the number of parallel training procedures executed at each iteration depends on the number of simultaneous activated cells.

7.3.2/ EXPERIMENTS OVERVIEW AND PARAMETERS

In this section, the parallel cellular model is mainly experimented with two set of tests: the first set of tests consists of a comparative evaluation between the GPU and the CPU applied on four disparity images of small sizes, the second set of tests concerns the application to larger size disparity maps in order to compare the performance of GPU and CPU as the image size increases. All the images used are from the stereo dataset of Middlebury [HS06], the four image pairs used in these tests ('Tsukuba', 'Venus', 'Teddy' and 'Cones') are illustrated in Fig. A.4 in Appendix A. The parameters which are kept constant for all tests on CPU and GPU are given in Table 7.2. For each of the two sets of tests, the supplemental parameters are presented in Table 7.3 and 7.4, respectively.

Table 7.2: EXPERIMENT PARAMETERS.

$MNPC$[1]	α_{init}	α_{final}	σ_{init}	σ_{final}	CPU_CRR[2]	GPU_CRR[3]
200	1	0.01	24	1	4 000	50

[1] Maximum Nodes Per Cell. [2] CPU_cell refresh rate. [3] GPU_cell refresh rate

In Table 7.3, are given the parameters for the first set of experiments. The neural network grid size, called mesh size, is obtained by $W/co \times H/co$. Coefficients co are set to 6, 4, 3, 3 for the four disparity images, respectively. Because of its size, the mesh can be considered as a compressed representation of the 3D surface defined by the disparity map. The cellular matrix sizes are obtained by $W/20 \times H/20$ for all images. It should be noted that the number of iterations on CPU corresponds to serial operations, and that the number of iterations on GPU corresponds to parallel operations. In Table 7.4, are given the parameters for the second set of experiments. The coefficient co for the three different image sizes are 3, 3, 6, respectively. The cellular matrix sizes are still obtained by $W/20 \times H/20$.

Table 7.3: EXPERIMENT PARAMETERS FOR COMPARATIVE EVALUATIONS ON CPU AND GPU.

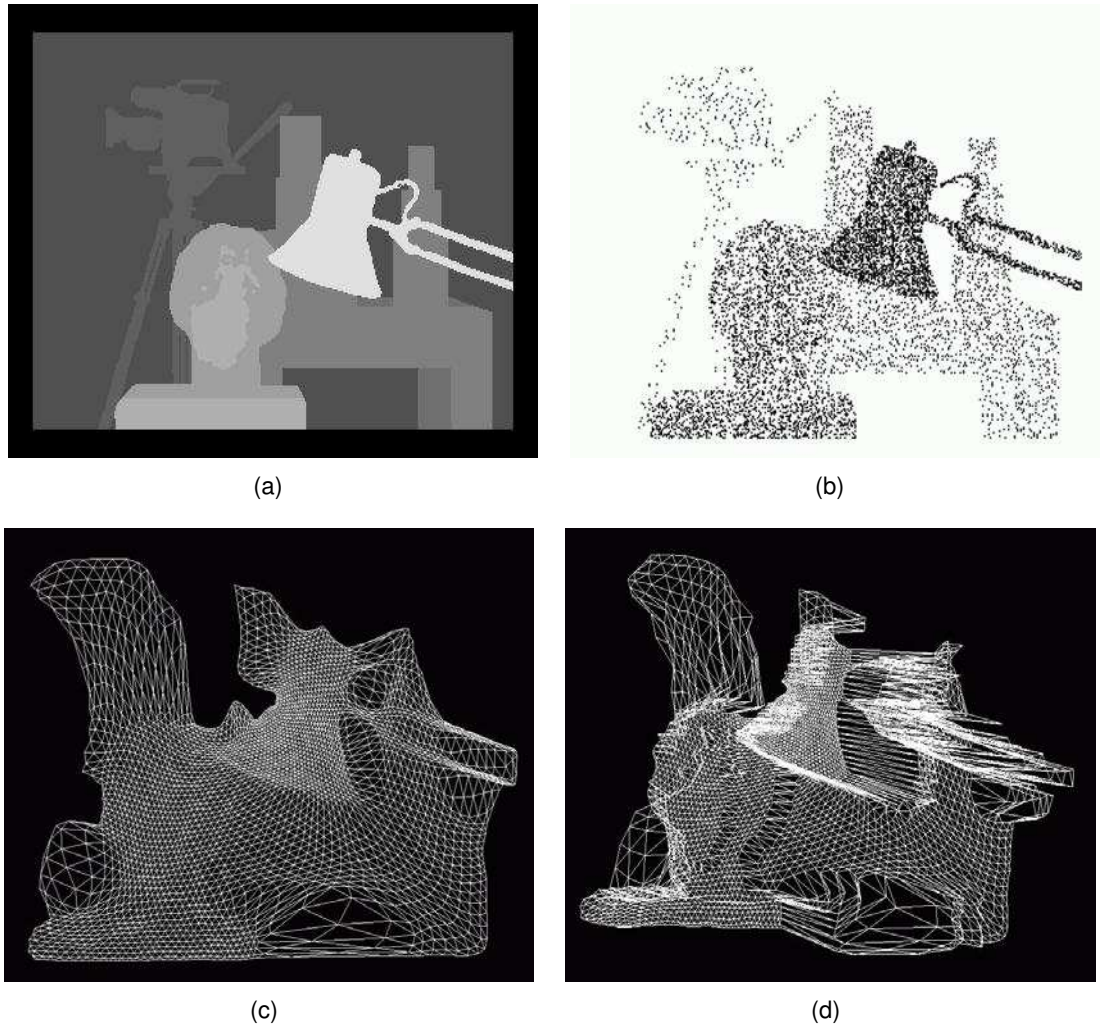
Image	Image size	grid size	cellular size	iteration_CPU	iteration_GPU
Tsukuba	384×288	64×48	20×15	1 000 000	1 500
Venus	434×383	110×96	22×20	1 000 000	1 500
Teddy	450×375	150×125	23×19	1 000 000	1 500
Cones	450×375	150×125	23×19	1 000 000	1 500

Table 7.4: EXPERIMENT PARAMETERS FOR LARGE SIZE IMAGES 'CONES'.

Image size	grid size	cellular size	iteration_CPU	iteration_GPU
450×375	150×125	23×19	1 000 000	1 500
900×750	300×250	45×38	1 000 000	1 500
1800×1500	300×250	90×75	2 000 000	1 500

7.3.3/ EXPERIMENTS RESULTS

In the first set of experiments on CPU and GPU, four input disparity images are considered with varying characteristics of sizes and textures. The comparative results are presented in Table 7.5 and two examples of visual results are presented on Fig. 7.2 and Fig. 7.3. The result values in the table are average values over five runs for each disparity image. The column 'Cost(%)' stands for the objective of the balanced structured mesh problem. It is the criteria that measures the adequacy of node density concentration between mesh and disparity map. The column 'CT(s)' stands for computation time measured in seconds. The parallel cellular model on GPU outperforms the serial model on computation time with similar cost minimization. Computation time takes about from 0.7 seconds to 1.2 seconds on GPU and from 6 seconds to 8 seconds on CPU. This leads to an acceleration factor of about 7 times faster in average. Besides the speedup, quality of results are roughly similar for both implantations.



(a) (b)

(c) (d)

Figure 7.2: Meshing of the 'Tsukuba' image. (a) Input disparity map. (b) Density sampling. (c) Meshing for Tsukuba image. (d) Visualization on 3D space.

The Fig. 7.2(a-c) illustrates a GPU result on the 'Tsukuba' image. In (a), is shown the

Table 7.5: MESHING EVALUATION RESULTS ON THE FOUR MIDDLEBURY DATA SETS OF SMALL SIZE.

		CPU		GPU	
Image	Size	Cost(%)	CT (s)	Cost(%)	CT (s)
Tsukuba	384×288	24.86	5.99	25.70	0.768
Venus	434×383	19.82	6.77	24.92	0.896
Teddy	450×375	22.52	7.48	23.14	1.2
Cones	450×375	19.30	7.86	22.6	1.18

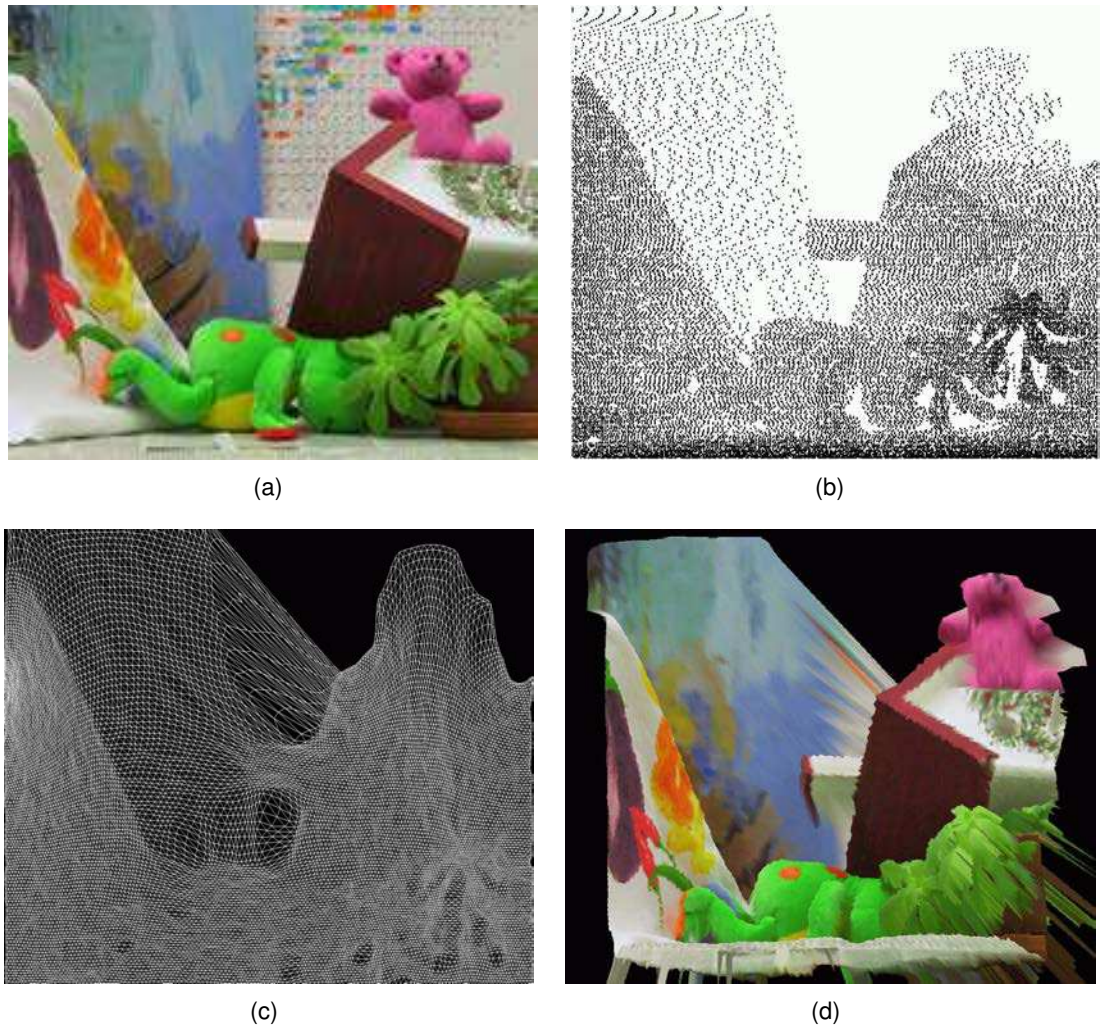
disparity map. In (b), is shown a sampling of the disparity map obtained by extraction of 10 000 points with a roulette wheel mechanism. This sample has been obtained after having removed the background small density values from the disparity map, and after augmenting the contrast in it. Then, objects that are closer to the camera have higher density values. In (c), is shown the adapted 2D mesh obtained by the GPU SOM algorithm applied to the modified disparity map. In Fig. 7.2(a), whiter regions are nearer to the camera view-point. In (b), such nearest regions present higher density of extracted points. In (c), the adapted grid presents higher density of neural network nodes on such regions, with respect of the topology of the scene. That is, proximity of grid points reflects proximity in Euclidean space.

A second illustration of the GPU meshing process is given in Fig. 7.3(a-d) on the 'Teddy' example. The image in (a) represents the left color stereo-image. A sampling of the disparity map, after removing background and augmenting contrast, is presented in (b), and the 2D adapted mesh in (c). The image in (d) presents the surface reconstruction in 3D space obtained by using the adapted mesh. Note that this mesh can be seen as a compressed representation of the 3D surface, such that objects closest to the camera have higher resolution and their details more finely represented. The meshing results of other two small size disparity maps, 'Venus' and 'Cones', are shown in Fig. 7.4.

Table 7.6: MESHING EVALUATION RESULTS ON LARGE SIZE VERSIONS OF 'CONES' IMAGE.

		CPU		GPU	
Image	Size	Cost(%)	CT (s)	Cost(%)	CT (s)
ConesT	450×375	19.30	7.86	22.6	1.18
ConesH	900×750	20.38	10.78	22.1	2.9
ConesF	1800×1500	17.50	24.14	15.38	9.3

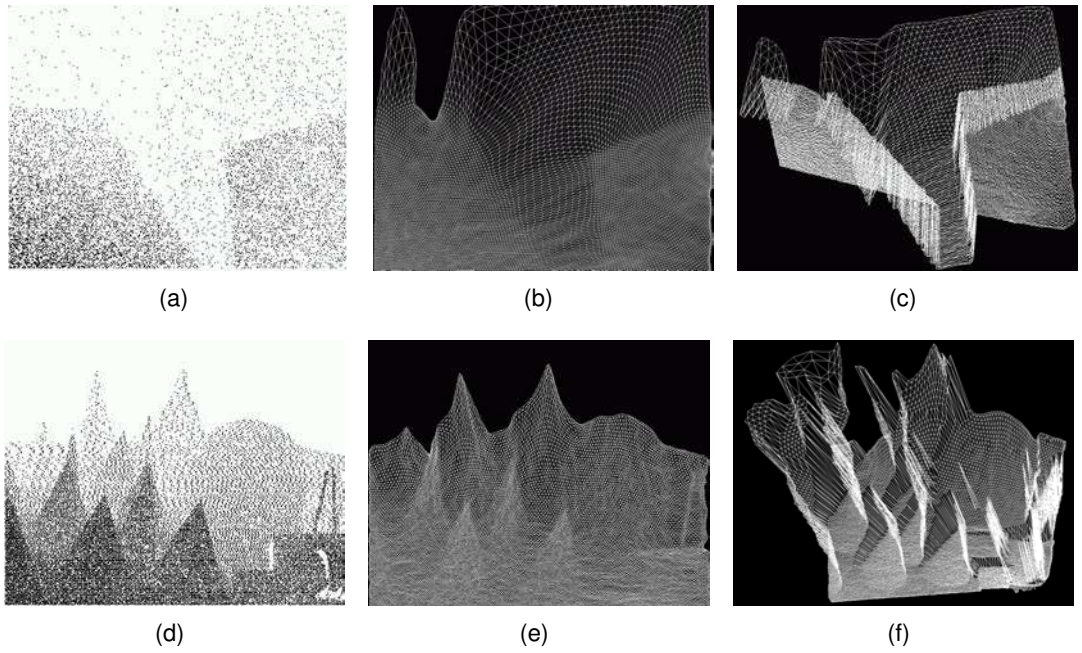
In the second set of experiments dealing with larger size disparity images, we use the Cones image to perform the comparative evaluation. The numerical results are given in Table 7.6 and resumed in Fig. 7.5. Here again, the GPU parallel cellular model outperforms its CPU counterpart in computation time for similar result quality. With the increase of image size, the performance in cost minimization of the GPU model augments substantially, whereas computation time slightly increases. In average, the GPU acceleration factor about 3 looks moderate. This indicates that it should be of interest to better understand and control warp divergence. We analyze that point in the next application.



(a) (b)

(c) (d)

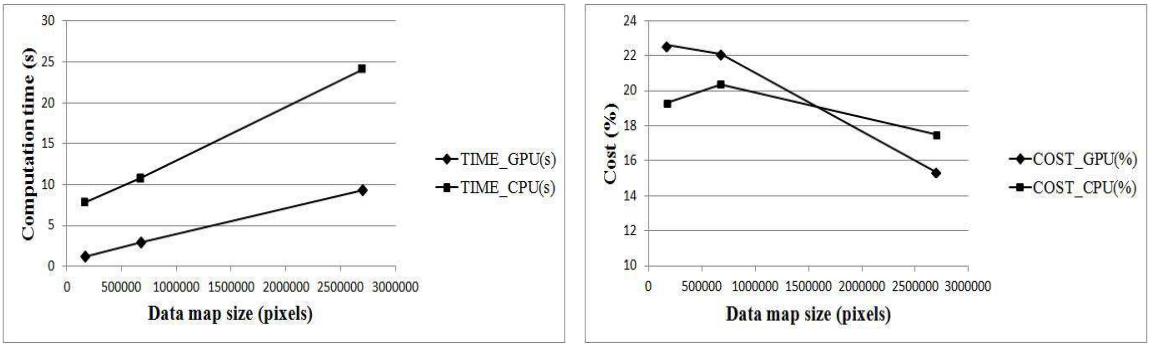
Figure 7.3: Meshing of 'Teddy' disparity map. (a) Original left color image. (b) Density sampling. (c) Meshing of Teddy image. (d) Visualization on 3D space.



(a) (b) (c)

(d) (e) (f)

Figure 7.4: Meshing of 'Venus' and 'Cones' disparity map. First column is density sampling. Second column is meshing result in 2D space. Third column is visualization on 3D space.



(a) (b)

Figure 7.5: Comparative evaluation between CPU and GPU for the increasing size 'Cones' images: (a) Comparison of relationship between computation time and image size in pixels. (b) Comparison of relationship between cost and image size in pixels.

7.4/ APPLICATION TO LARGE SCALE EUCLIDEAN TSP

7.4.1/ WARP DIVERGENCE ANALYSIS

In the CUDA architecture, a warp refers to a collection of 32 threads that are "woven together" and get executed in lockstep [NVI12b]. At every line in kernel function, each thread in a warp executes the same instruction on different data. When some of the threads in a warp need to execute an instruction while others in the same warp do not, this situation is known as warp divergence or thread divergence. Under normal circumstances, divergent branches simply result in performance degradation, with some threads remaining idle while the other threads actually execute the instructions in the branches. The execution of threads in a warp with divergent branches are therefore carried out sequentially, resulting in performance degradation.

Name	Grid Size	Block Size	Branch Efficiency	branch	divergent branch
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	90.1%	3577	434
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	90%	3200	389
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.9%	4593	465
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.8%	3591	383
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.6%	4108	415
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.6%	3895	397
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.4%	3609	368
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.3%	3134	454
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89.1%	3281	402
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	89%	3173	435

(a)

Name	Grid Size	Block Size	Branch Efficiency	branch	divergent branch
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	87.2%	7065	835
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	87.2%	7518	982
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	87.2%	7194	1047
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.7%	7513	987
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.7%	6685	1067
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.4%	6635	1039
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.4%	8040	917
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.3%	6612	944
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.2%	7187	1038
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	86.2%	7365	994

(b)

Name	Grid Size	Block Size	Branch Efficiency	branch	divergent branch
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.9%	10815	1704
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.7%	11537	1631
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.6%	11675	1748
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.4%	11239	1791
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.3%	9797	1664
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85.1%	9944	1469
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	85%	10261	1255
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	84.9%	10175	1686
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	84.9%	10355	1666
gpu_SpiralSearch_Learning_tsp	[2,2,1]	[16,16,1]	84.9%	11034	1848

(c)

Name	Grid Size	Block Size	Branch Efficiency	branch	divergent branch
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96.9%	9336	434
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96.5%	11617	412
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96.3%	8788	334
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96.1%	9909	425
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96%	8544	431
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	96%	8894	327
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	95.9%	9485	380
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	95.8%	10573	325
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	95.8%	8700	361
gpu_SpiralSearch_Learning_tsp	[12,6,1]	[2,4,1]	95.8%	8113	350

(d)

Figure 7.6: Branch Efficiency with different search ranges.

According to our trial tests, the most time consuming kernel function is parallel SOM. One of the reasons is that there exists warp divergence when this kernel is being executed, because it has an unpredictable spiral search process in it. The spiral search is carried out in each cell of the search range, one by one, and it stops immediately when the thread finds a nearest neuron. As a result, different threads may stop at different times. Also, the more cells each thread is going to search in, the severer this problem gets. Hence, different search range settings have different influences on warp divergence. When the block size is set to 256, which is usually enough to fulfill the streaming multi-processor with adequate warps for the GPU device with CUDA capability 2.0, the highest branch efficiencies

(ratio of non-divergent branches to total branches [NVI12a]) of all executions with search range of 1, 2, and 3 are 90.1%, 87.2%, and 85.9%, respectively, as collected by NVIDIA Visual Profiler, shown in Fig. 7.6(a-c). In theory, the less threads are put in one block, the less warp divergence occurrences will appear. Extremely, if there is only one thread in a block, then, there will definitely not be warp divergence. However, the decrease of threads in each block implies the decrease of the CUDA cores usage associated to each streaming multi-processor. In order to analyze the trade-off between performance and number of thread in a block, we have tested a set of different combinations of grid size and block size for the SOM kernel. The configuration, which makes the kernel run fastest, is with block size of 8 with highest branch efficiency of 96.9%.

7.4.2/ EXPERIMENTS OVERVIEW AND PARAMETERS

Table 7.7: EXPERIMENT PARAMETERS.

	α_{init}	α_{final}	σ_{init}	σ_{final}	iterations	δ	CRR[a]	SSR[b]	MRSR[c]
GPU[1]	1	0.01	12	1	100000	1	1	1	1000
CPU[1]	1	0.01	12	1	$100000 \times N$	–	100	1	–
GPU[2]	1	0.01	100	1	100000	1	1	3	1000
CPU[2]	1	0.01	100	1	$10000 \times N$	–	100	3	–

[1] Tests of small size instances. [2] Tests of large size instances.

[a] Cell refresh rate. [b] Spiral search range. [c] Memory reuse set rate.

We have done our tests with two groups of instances from either National TSPs [1] and TSPLIB database [Rei91]. One group consists of four small size instances from 124 cities to 980 cities, while the other consists of four large size instances from 8246 cities to 33708 cities. The parameter settings for the two groups are shown in Table 7.7. As discussed in Section 5.4, $T_{max} \times \lceil \sqrt{N \times \lambda} \rceil^2$ parallel SOM operations will be carried out as an extreme case by the GPU SOM program, with N the input instance size and λ set to 1.1. For the tests of small size instances, we set the total number of sequential iterations of the CPU version to $T_{max} \times N$, in order to make the total SOM operations approximately similar between GPU version and CPU version, and to reach similar quality results. Whereas for the tests with large size instances, we set it to $T_{max} \times N/10$, also to achieve similar quality results and because GPU operations depend on the cell activation probabilities and may be less than N at each GPU parallel iteration.

7.4.3/ COMPARATIVE GPU/CPU RESULTS ON LARGE SIZE TSP PROBLEMS

All the tests are done on a basis of 10 runs per instance. For each test case is reported the percentage deviation, called "%PDM", to the optimum tour length of the mean solution value obtained, i.e. $\%PDM = (mean\ \ length - optimum) \times 100/optimum$. As well, is reported the percentage deviation from the optimum of the best solution value found over 10 runs, called "%PDB". Finally, is also reported the average computation time per run in seconds, called "Sec".

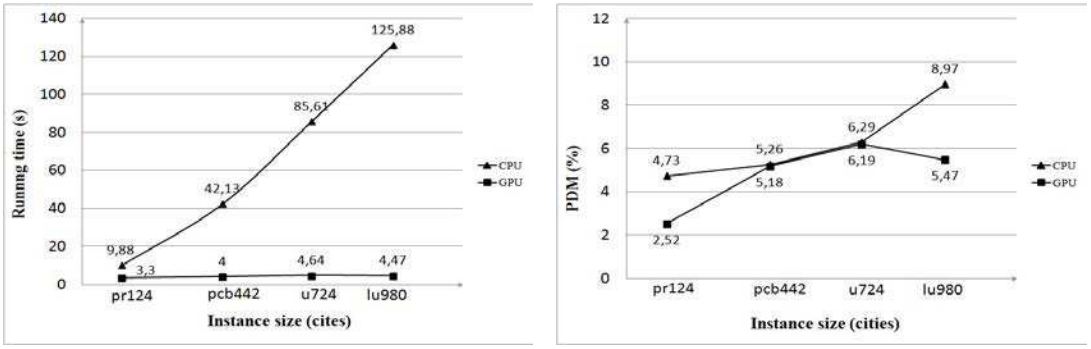
[1] Refer to http://www.math.uwaterloo.ca/tsp/world/countries.html

Table 7.8: TEST RESULTS ON SMALL SIZE INSTANCES.

		GPU			CPU		
Problem	Optimal	%PDM	%PDB	Sec	%PDM	%PDB	Sec
pr124	59030	2.52	1.07	3.30	4.73	1.85	9.88
pcb442	50778	5.18	3.41	4.00	5.26	3.24	42.13
u724	41910	6.19	4.96	4.64	6.29	4.67	85.61
lu980	11340	5.47	3.40	4.47	8.97	4.58	125.88
Average		4.84	3.21	4.10	6.31	3.59	65.88

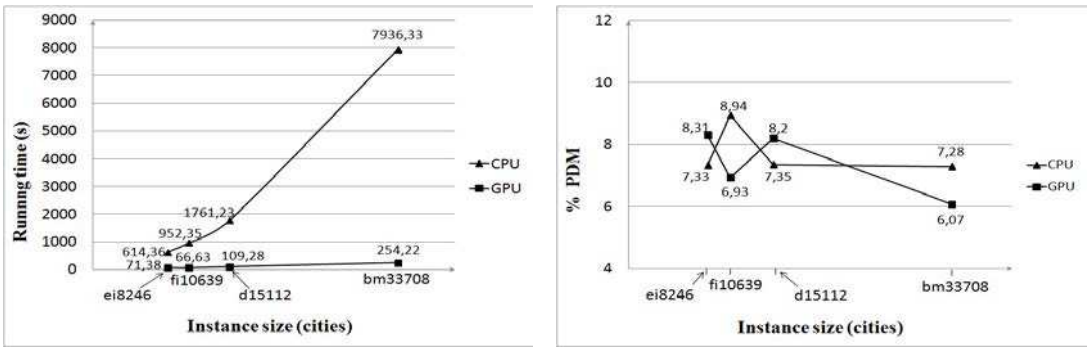
Table 7.9: TEST RESULTS ON LARGE SIZE INSTANCES.

		GPU			CPU		
Problem	Optimal	%PDM	%PDB	Sec	%PDM	%PDB	Sec
ei8246	206171	8.31	7.12	71.38	7.33	6.88	614.36
fi10639	520527	6.93	6.49	66.63	8.94	8.10	952.35
d15112	1573084	8.20	7.66	109.28	7.35	7.14	1761.23
bm33708	959304	6.07	5.85	254.22	7.28	7.04	7936.33
Average		7.38	6.78	125.38	7.73	7.29	2816.07



(a) (b)

Figure 7.7: Test results of small size instances.



(a) (b)

Figure 7.8: Test results of large size instances.

As shown in Fig. 7.7 and Fig. 7.8, and in Table 7.8 and Table 7.9, respectively, for the two instance groups, our GPU parallel SOM approach outperforms its counterpart CPU sequential version both on small size and large size instances, for similar tour length results. For small size instances, the ratio of CPU time by GPU time (called acceleration factor) varies from roughly factor 3 to factor 28, as the instance size grows. For large size instances, it varies from roughly factor 9 to factor 31 for the maximum size instance with up to 33708 cities. We think that the acceleration factor augmentation indicates a better streaming multi-processor occupancy as the instance size grows. We can note that the execution time of GPU version increases in a linear way with a very weak increasing coefficient, when compared to the CPU version execution time. We consider that such results are encouraging in that the parallel SOM model should really exploit the benefits of multi-processors, as the number of physical cores will augment in the future.

7.5/ CONCLUSION

In this chapter, we have presented a CUDA implemention of the cellular GPU parallel model for self-organizing map and its application to structured mesh generation and traveling salesman problem. The structured mesh generation problem is presented as a NP-hard balanced clustering problem in the plane, and the SOM process is used as an heuristic to deal with it. In the same way, we have presented the CUDA implementation of self-organizing map and application to large scale Euclidean traveling salesman problems with up to 33708 cities. While the GPU implantation for meshing was roughly three time faster than CPU, we get an acceleration of about factor 30 for TSP GPU implementation. Problem of warp divergence looks prominent in first application, while it has less impact in second application. We should more closely studied warp divergence in the future to better exploit warp parallelism that manages threads into blocks. Test results show that our GPU model has linear increasing execution time with a very weak increasing coefficient when compared to the CPU version, for both small size instances and large size instances. The theoretical computation time of our model is based on a parallel execution of many spiral search of closest points, each one having a time complexity in $O(1)$ in average when dealing with a uniform, or at most a bounded data distribution. We would like to reach the theoretical computation acceleration factor N, with N the problem size. But when trying solving a Euclidean NP-hard optimization problem, the price to pay comes from the many concurrent accesses and interactions, together with memory uncoalescing and warp divergence, that clearly introduce serialization into the parallel computation. However, actual acceleration factors obtained look promising and will help define in the future the best adequate relationship between the abstract computation model and the physical multi-core platform.

III

Conclusions and Perspectives

8

CONCLUSION

8.1/ GENERAL CONCLUSION

We have presented some parallel GPU models that allow to improve the effectiveness in computation for optimization problems defined in the Euclidean space. One difficulty is finding effective parallel computation strategies that match the computation capacity of GPU architectures. We presented the cellular GPU parallel model as a way of looking at problem resolution of spatially distributed problems, where data stand for entities in the plane or 3D space. The repetition of many locally distributed operations is the basis for parallel computation of the solution. The goal was to allow both dealing with large size problems and provide substantial acceleration factors by massive parallelism. The main characteristic of the cellular model is that it decomposes the plane into an appropriate number of cellular units, each responsible of a constant part of the input data. Each cell corresponds to a single processing unit. However, the exploitation of parallel model is not trivial and many issues related to the GPU memory hierarchical management and its parallel processing architecture have to be considered. Certainly, the many concurrent accesses and interactions, together with memory uncoalescing and warp divergence due to data distribution dependence, clearly introduce serialization into the parallel computation. Finding the best adequate relationship between abstract computation model and physical multi-core platform remains a challenge.

The main characteristics of the cellular parallel GPU model proposed are summarized hereafter:

- Distributed solution computation from scratch (no need of a construction method) and no central control.
- Low granularity level based on data decomposition.
- Linear association of GPU resources to input data for large problem size handling.
- Extensive experiments according to problem size.
- Systematic report of solution quality and computation time.

Applications presented in this document should help clarify the benefit of GPU usage on some complex but naturally distributed optimization problems. They are from two interconnected domains: image stereo-matching domain, where each parallel processing unit naturally corresponds to a single pixel of the image, and structured mesh generation by using the self-organizing map neural network, where each processing unit represents

a local area of the plane with its data. Application to the standard Euclidean traveling salesman problem completes the evaluation. More precisely, the document presented four GPU applications. The first application concerns a stereo-matching problem for color stereovision. Taking an image pair as input, the output is a disparity map that represents depths in the 3D scene. The goal was to implement and compare GPU/CPU versions of a winner-takes-all local dense stereo-matching method dealing with CFA (color filter array) image pairs. It is clear that the method naturally fits into the cellular approach. While a simple and naïve GPU implementation can accelerate the CPU counterpart, this naive method was not able to guarantee real-time requirement. The GPU speedup factor over CPU was of 20 times faster for the CFA image pairs, but computation time was about 0.2s for a small image pair. The second application focused on the possible GPU improvements to reach near real-time stereo-matching computation. By a better memory management, the use of adaptive windows and specific filters, the near real-time parallel stereovision algorithm takes about 0.017s for a small image pairs. Computation times obtained correspond to ones of the fastest records of the Middlebury benchmark with moderate solution quality. Several GPU versions were considered, and the results were also compared to a CPU dynamic sequential programming approach.

The third and fourth applications deal with a cellular GPU implementation of the self-organizing map in the plane. The third problem makes a relationship to the stereo-matching application by addressing mesh generation according to a disparity map. Mesh generation allows 3D surface visualization and compressed representation of topology. The approach was tested on Middlebury database to gauge the GPU acceleration factor and quality obtained. The last application studied was the Euclidean traveling salesman problem (TSP) with large scale instances with up to 33708 cities. The mesh is now a ring structure that matches cities. In both applications, GPU implementations allow substantial acceleration factors over CPU versions, as the problem size increases and for similar quality results. In a general manner, we evaluated the effectiveness of our cellular parallel model through extensive experiments, systematically reporting quality and computation time according to problem instance size. At the moment of writing, we have not found in the literature GPU implementations able to address such large size TSP instances in GPU. We think that this is because current GPU optimization applications to the TSP are memory consuming algorithms, such as ant colony, genetic algorithm or k-opt local search. The acceleration factor for the GPU parallel self-organizing map over the CPU version on the largest TSP problem with 33708 cities was 30 times faster. We think that such acceleration factor is substantial and encouraging with regards to other approaches. Most often, applications compare CPU and GPU versions of a given algorithm that strictly match the semantic of the sequential version. This guarantees equivalence of behavior of the two algorithms and authors simply report the acceleration factors. Here, we also report solution quality. This will help compare different approaches when dealing with non standard methods. Note that the most powerful sequential heuristics for TSP use complex data structures and specific implementation tricks that enhance and modify the standard local search technique.

8.2/ PERSPECTIVE AND FURTHER RESEARCH DIRECTIONS

Several axis can be considered as extensions of this work. They can address a better use of GPU resources, a better understanding of the adequate GPU implementation struc-

ture, application and generalization to other optimization problems, extension to other techniques, and use of other parallel platforms.

Future work should deal with verification of effectiveness of the algorithm as the number of physical cores augments. For example, we should more closely study the impact of the number of GPU streaming multi-processors or simple cores. It should be of interest also to develop techniques for a better memory coalescing, better use of memory and combination with shared memory. Improving the balance between processor loads, is of interest for reducing warp divergence. Since processor load depends on data distribution, it would be interesting to better study its impact on computation time. A solution for balancing the load should be to adapt the cellular decomposition of the data to its distribution in the plane. Balanced structured meshing could be in help here, by generating a cellular decomposition where cells share an equal quantity of data, at the condition where such cellular decomposition is used for some subsequent problem.

Many algorithms can be developed based on an underlying parallel spiral search framework for closest point findings. A simple application is Voronoï tesselation computation and Delaunay triangulation. A well-known result is $O(N)$ computation time for Voronoï tessellation according to spiral search for uniform or bounded distributions of size N. We should investigate such algorithms for mesh generation in general, even for unstructured mesh generation approaches.

Extension to other NP-hard optimization problems can also be considered. Cellular parallel model could help the design of specific GPU operators embedded into more general methods such as population based metaheuristics. Application of SOM to different types of vehicle routing problems already exist and acceleration methods could be obtained by GPU implantations of SOM like operators. Combination with local search operators could also be addressed in this way. Extending the cellular model for dealing with standard neighborhood operations could be considered in the future. Rather than executing a given step of a sequential local search in parallel, it could be envisaged to perform some parallel distributed search where each processor would be responsible for a given local part of the data and local improvement moves. Threads would be in competition to improve a shared global objective function composed of many local components.

Because of the different granularity levels of parallelization, it is also of interest to study hybrid combination of parallel systems at different levels. Coarse grain parallelism, at the level of solution duplication, as in population based search, should be exploited in clusters or standard networks of workstations, whereas fine grain parallelism at the level of solution decomposition should be exploited on massive GPU parallel systems operating on global and shared memory. So, when dealing with such problems in which the computations become asynchronous, using cluster of workstations or computational grids might be more relevant.

IV

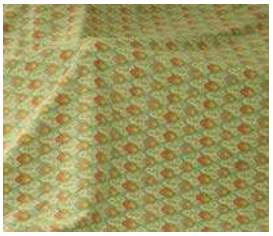
APPENDIX

A

Input image pairs

(a) Aloe left image

(b) Bowling1 left image



(c) Cloth1 left image

(d) Flowerpots left image

(e) Lampshade1 left image

(f) Midd1 left image

(g) Monopoly left image

(h) Plastic left image

(i) Rock1 left image

(j) Wood1 left image

Figure A.1: Left image of each tested stereo image pair from Middlebury database.

(a) Aloe right image

(b) Bowling1 right image

(c) Cloth1 right image

(d) Flowerpots right image

(e) Lampshade1 right image

(f) Midd1 right image

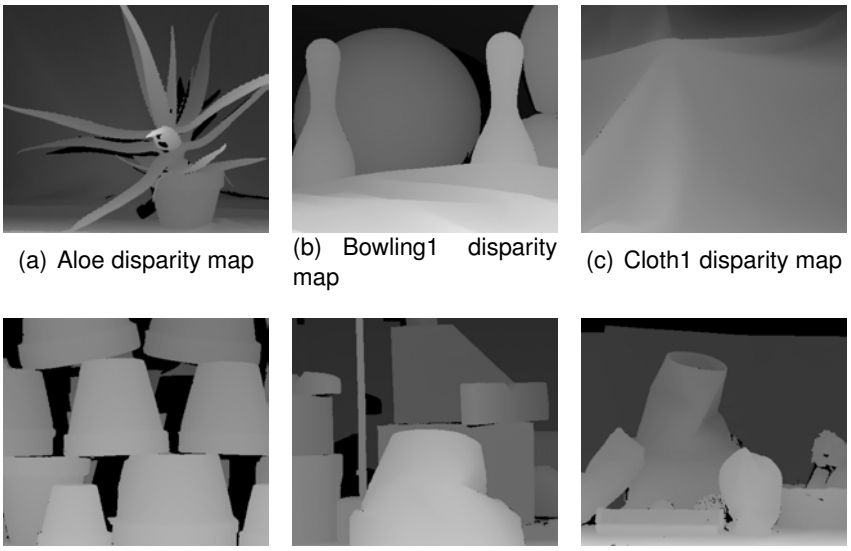
(g) Monopoly right image

(h) Plastic right image

(i) Rock1 right image

(j) Wood1 right image

Figure A.2: Right image of each tested stereo image pair from Middlebury database.



(a) Aloe disparity map

(b) Bowling1 disparity map

(c) Cloth1 disparity map

(d) Flowerpots disparity map

(e) Lampshade1 disparity map

(f) Midd1 disparity map

(g) Monopoly disparity map

(h) Plastic disparity map

(i) Rock1 disparity map

(j) Wood1 disparity map

Figure A.3: Benchmark disparity map of each tested stereo image pair from Middlebury database.

(a) Rock1 left image (b) Rock1 right image (c) Rock1 true disparity map

(d) Aloe left image (e) Aloe right image (f) Aloe true disparity map

(g) Tsukuba left image (h) Tsukuba right image (i) Tsukuba true disparity map

(j) Cones left image (k) Cones right image (l) Cones true disparity map

(m) Teddy left image (n) Teddy right image (o) Teddy true disparity map

(p) Venus left image (q) Venus right image (r) Venus true disparity map

Figure A.4: Tested image pairs from Middlebury database.

B

EXPERIMENT RESULTS

B.1/ CFA STEREO MATCHING RESULTS ON FULL SIZE IMAGES

B.2/ CFA STEREO MATCHING RESULTS ON HALF SIZE IMAGES

B.3/ CFA STEREO MATCHING RESULTS ON SMALL SIZE IMAGES

Table B.1: THE TABLE OF COMPUTATION-TIME (S) (CT(S)) AND THE RCMP (%) OF THE DATASETS OF FULL SIZE IMAGES IN THE EXPERIMENTS.

Image group	Plate form		Half-window 1	2	3	4	5	6	7	8	9	10
Aloe	CPU	CT	453	1196	2274	3681	5449	7552	10024	12793	15879	19290
		RCMP	49,17	62,19	67,69	69,76	70,63	71,01	71,06	70,97	70,76	70,47
	GPU	CT	22	47	100	158	248	398	570	706	827	915
		RCMP	48,35	59,69	64,92	69,26	68,22	67,23	66,94	66,44	65,79	65,44
Bowling1	CPU	CT	434	1145	2181	3531	5210	7223	9579	12221	15164	18414
		RCMP	9,46	12,52	17,20	21,44	24,89	27,65	29,85	31,66	33,21	34,57
	GPU	CT	23	51	113	167	224	417	527	698	793	934
		RCMP	9,38	12,30	16,85	21,01	24,40	27,17	29,33	31,17	32,72	34,02
Cloth1	CPU	CT	433	1141	2169	3511	5180	7183	9534	12140	15071	18311
		RCMP	50,34	68,15	79,26	84,41	86,60	87,54	87,96	88,11	88,10	88,01
	GPU	CT	22	56	97	170	274	417	515	645	821	892
		RCMP	49,73	67,25	78,53	83,84	86,23	86,35	86,72	86,75	86,63	86,44
Flowerpots	CPU	CT	475	1254	2384	3853	5686	7885	10483	13349	16585	20144
		RCMP	8,11	8,80	12,71	16,70	20,32	23,53	26,43	28,94	31,12	33,00
	GPU	CT	23	61	111	154	240	398	540	756	836	928
		RCMP	7,95	8,65	12,56	16,56	18,77	67,23	23,87	25,87	27,58	29,01
Lampshade1	CPU	CT	474	1250	2377	3851	5679	7854	10406	13273	16498	20021
		RCMP	8,16	7,75	10,28	13,40	16,53	19,51	22,18	24,50	26,56	28,37
	GPU	CT	22	51	111	204	291	398	546	774	859	900
		RCMP	8,02	7,57	10,12	13,37	15,46	15,43	15,76	16,61	17,90	18,99
Midd1	CPU	CT	540	1423	2702	4381	6463	8968	11899	15184	18870	22925
		RCMP	12,72	17,41	21,77	24,92	27,11	28,71	29,91	30,86	31,64	32,15
	GPU	CT	27	63	118	193	283	476	573	737	925	1006
		RCMP	12,33	16,57	20,65	23,78	21,43	22,18	22,76	23,10	23,32	23,45
Monopoly	CPU	CT	488	1285	2449	3957	5840	8103	10749	13711	17042	20715
		RCMP	11,73	17,53	22,60	26,84	30,26	33,12	35,58	37,62	39,24	40,55
	GPU	CT	21	54	97	183	290	423	512	724	870	950
		RCMP	11,49	16,91	21,68	25,80	23,29	24,90	25,26	26,61	27,97	28,45
Plastic	CPU	CT	449	1183	2251	3643	5372	7457	9878	12592	15636	18992
		RCMP	5,20	5,79	7,00	8,26	9,49	10,65	11,73	12,77	13,76	14,67
	GPU	CT	21	63	105	167	233	378	527	666	809	908
		RCMP	5,15	5,70	6,90	7,91	9,11	10,15	10,83	11,97	12,76	14,01
Rocks1	CPU	CT	449	1184	2250	3643	5374	7453	9911	12609	15675	19027
		RCMP	25,44	39,61	51,68	57,86	60,94	62,54	63,42	63,90	64,23	64,37
	GPU	CT	20	46	90	171	237	391	575	685	764	896
		RCMP	24,89	38,54	50,85	57,35	60,66	60,52	61,04	61,17	62,67	60,90
Wood1	CPU	CT	520	1371	2608	4222	6239	8655	11472	14637	18175	22098
		RCMP	15,79	18,38	26,43	34,75	41,60	47,11	51,43	54,90	57,55	59,64
	GPU	CT	21	73	117	194	281	457	573	732	911	991
		RCMP	15,40	17,88	25,82	34,14	36,98	41,62	45,26	48,10	50,35	52,15
Disparity error tolerance δ = 1												

Table B.2: THE TABLE OF COMPUTATION-TIME (S) (CT(S)) AND THE RCMP (%) OF THE DATASETS OF HALF SIZE IMAGES IN THE EXPERIMENTS.

Image group	Plate form		Half-window									
			1	2	3	4	5	6	7	8	9	10
Aloe	CPU	CT	57	148	281	453	666	917	1205	1531	1893	2288
		RCMP	59,66	69,27	71,75	72,40	72,41	72,03	71,46	70,82	70,21	69,71
	GPU	CT	11	38	72	118	174	233	312	409	487	527
		RCMP	56,37	66,46	69,44	70,31	70,30	69,74	69,09	68,37	67,72	67,03
Bowling1	CPU	CT	54	142	269	434	638	878	1154	1465	1809	2186
		RCMP	16,28	21,87	28,49	33,71	37,71	40,93	43,65	45,88	47,70	48,98
	GPU	CT	13	36	54	95	157	231	298	421	499	565
		RCMP	14,91	20,50	27,23	32,12	36,17	39,32	41,79	43,90	45,62	47,12
Cloth1	CPU	CT	54	141	267	432	633	872	1147	1455	1797	2172
		RCMP	61,24	78,92	85,43	87,16	87,75	87,91	87,93	87,91	87,80	87,64
	GPU	CT	14	38	69	105	169	211	280	367	500	534
		RCMP	58,71	76,81	84,29	86,73	87,77	88,25	88,46	88,60	88,69	88,72
Flowerpots	CPU	CT	59	155	294	475	698	961	1264	1604	1982	2396
		RCMP	15,86	19,64	26,72	32,95	37,85	41,50	44,37	46,45	47,98	49,08
	GPU	CT	15	26	35	109	158	234	314	379	435	597
		RCMP	12,92	13,81	16,04	17,36	18,56	19,47	20,36	21,02	21,59	21,98
Lampshade1	CPU	CT	59	154	292	471	692	953	1253	1591	1966	2375
		RCMP	14,00	15,57	20,80	26,28	30,73	34,08	36,60	38,55	39,99	41,29
	GPU	CT	15	34	70	121	161	222	299	391	436	516
		RCMP	11,96	11,61	13,29	15,14	16,71	17,76	18,31	18,47	18,47	18,34
Midd1	CPU	CT	67	177	335	541	794	1094	1439	1828	2260	2732
		RCMP	20,55	26,71	31,80	34,93	36,76	37,92	38,50	38,76	38,89	38,91
	GPU	CT	14	39	65	120	195	214	324	414	506	586
		RCMP	19,24	24,99	30,00	33,25	35,23	36,52	37,40	37,81	37,98	37,90
Monopoly	CPU	CT	61	160	302	488	716	986	1298	1647	2036	2461
		RCMP	20,92	29,12	35,22	39,52	42,41	44,61	46,26	47,60	48,83	49,81
	GPU	CT	15	40	74	124	188	258	347	403	476	682
		RCMP	16,07	21,90	24,73	27,92	29,90	31,50	32,79	33,80	34,66	35,19
Plastic	CPU	CT	56	147	278	448	659	907	1193	1515	1871	2261
		RCMP	9,39	11,39	13,98	16,43	18,79	20,87	22,67	24,16	25,44	26,58
	GPU	CT	14	39	58	115	168	246	305	348	399	571
		RCMP	8,18	9,92	12,05	14,26	16,40	18,30	20,11	21,83	23,32	24,62
Rocks1	CPU	CT	56	147	278	449	659	907	1194	1515	1872	2263
		RCMP	36,94	53,26	63,43	67,84	70,00	71,15	71,70	72,02	72,17	72,14
	GPU	CT	15	40	69	115	164	226	300	359	498	527
		RCMP	35,45	51,43	62,42	67,49	69,89	71,21	72,08	72,67	72,98	73,17
Wood1	CPU	CT	65	170	323	522	766	1056	1390	1766	2183	2640
		RCMP	24,82	30,77	39,16	46,62	52,84	57,79	61,52	64,19	65,99	67,19
	GPU	CT	15	40	68	114	179	218	305	394	517	540
		RCMP	22,90	29,31	38,17	45,58	51,33	55,84	59,33	61,84	63,43	64,40
Disparity error tolerance δ = 1												

Table B.3: THE TABLE OF COMPUTATION-TIME (S) (CT(S)) AND THE RCMP (%) OF THE DATASETS OF SMALL SIZE IMAGES IN THE EXPERIMENTS.

Image group	Plate form		Half-window 1	2	3	4	5	6	7	8	9	10
Aloe	CPU	CT	16	43	81	130	190	261	342	432	531	638
		RCMP	60,70	69,88	71,73	71,80	71,38	70,78	70,17	69,63	69,08	68,48
	GPU	CT	11	28	45	80	119	139	189	265	326	407
		RCMP	56,66	66,72	69,79	70,50	70,57	70,38	70,16	69,03	68,96	68,38
Bowling1	CPU	CT	15	41	77	125	183	250	328	414	508	611
		RCMP	18,93	25,42	32,95	39,13	44,06	47,91	50,88	52,77	53,90	54,51
	GPU	CT	6	25	42	74	106	137	172	252	332	406
		RCMP	16,07	20,51	25,21	28,80	32,01	34,69	36,53	37,65	37,39	36,91
Cloth1	CPU	CT	15	41	77	124	181	249	326	411	505	607
		RCMP	58,69	77,10	84,29	86,47	87,19	87,50	87,65	87,66	87,56	87,31
	GPU	CT	9	25	43	82	109	144	195	230	326	365
		RCMP	56,53	75,63	84,10	87,34	88,98	90,03	90,74	91,25	91,70	92,14
Flowerpots	CPU	CT	17	45	85	137	200	274	359	454	558	672
		RCMP	21,31	26,86	34,91	41,35	46,04	49,30	51,46	52,74	53,66	54,03
	GPU	CT	8	22	50	85	123	158	196	241	357	433
		RCMP	17,82	21,63	27,53	32,23	35,43	37,31	38,65	39,48	40,13	40,39
Lampshade1	CPU	CT	17	45	84	136	198	272	356	450	553	665
		RCMP	17,72	19,14	24,11	28,68	32,43	35,43	37,71	39,58	41,16	42,35
	GPU	CT	9	24	49	80	130	145	215	302	258	394
		RCMP	15,29	16,11	19,51	22,14	24,51	26,43	27,82	29,17	30,19	30,80
Midd1	CPU	CT	19	51	97	156	228	313	409	517	636	766
		RCMP	24,66	31,12	35,81	38,47	39,82	40,61	40,93	41,17	41,33	41,34
	GPU	CT	12	30	64	95	130	193	235	278	371	426
		RCMP	22,89	29,27	34,06	37,03	38,77	39,75	40,17	40,19	40,21	40,02
Monopoly	CPU	CT	17	46	87	141	206	282	369	467	574	690
		RCMP	26,06	34,61	40,71	44,80	47,64	49,71	51,44	52,69	53,66	54,55
	GPU	CT	10	23	50	74	109	167	204	225	364	425
		RCMP	21,75	29,55	34,43	37,11	38,64	39,53	40,27	40,52	40,77	40,89
Plastic	CPU	CT	16	42	80	129	189	259	338	428	525	632
		RCMP	12,71	15,44	18,70	21,66	24,36	26,90	28,88	30,35	31,73	32,98
	GPU	CT	9	24	45	70	115	151	194	245	322	377
		RCMP	10,36	11,91	13,87	15,32	16,88	18,29	19,47	20,30	21,20	21,84
Rocks1	CPU	CT	16	42	80	129	189	259	339	428	526	633
		RCMP	42,95	59,04	68,07	71,78	73,49	74,11	74,23	74,22	73,94	73,59
	GPU	CT	10	26	45	74	111	150	177	240	304	342
		RCMP	40,24	56,36	66,48	71,01	73,61	75,09	75,90	76,39	76,70	76,86
Wood1	CPU	CT	19	49	93	150	220	302	395	499	614	738
		RCMP	27,79	33,85	42,34	49,52	54,91	59,07	62,24	64,59	66,01	67,69
	GPU	CT	10	29	49	79	124	183	215	290	363	427
		RCMP	25,80	32,08	40,56	48,17	54,15	58,53	61,63	64,08	65,93	67,24
Disparity error tolerance δ = 1												

List of Figures

List of Tables

BIBLIOGRAPHY

[BBh03] M. Z. Brown, Z. Burschka, and G. D. hager. Advances in computational stereo. . *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 993–1008, August 2003.

[BEP05] F. Boudjemai, P. B. Enberg, and J. G. Postaire. Dynamic adaptation and subdivision in 3D-SOM application to surface reconstruction. In *Tools with Artificial Intelligence, 2005. ICTAI 05. 17th IEEE International Conference on*, pages 6–pp. IEEE, 2005.

[BGMT08] S. Battiato, M. Guarnera, G. Messina, and V. Tomaselli. Recent patents on color demosaicing. *Recent Patents on Computer Science*, pages 1(3):194–207, November 2008.

[BHP01] R. Baraglia, J. I. Hidalgo, and R. Perego. A Parallel Hybrid Heuristic for the TSP. *Application of Evolutinaey Computing Lecture Notes in Computer Science*, 2037:193–202, 2001.

[BI99] A. F. Bobick and S. S. Intille. Large occlusion stereo. *International Journal of Computer Vision*, pages 181–200, 1999.

[BK99] D. Beymer and K. Konolige. Real-Time Tracking of Multuiple people Using Continuous Detection. *IEEE Frame Rate Workshop*, 1999.

[Bon89] E. Bonomi. Generation of structured adaptative grids based upon molecular dynamics. *Comptes rendus des journées d'électronique, Presses Polytechniques Romandes, Lausanne*, 1989.

[Bor06] P. Borovska. Solving the Travelling Salesman Problem in Parallel by Genetic Algorithm on Multicomputer Cluster. *International Conference on Computer Systems and Technologies*, 2006.

[BT98] S. Birchfield and C. Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 401–406, 1998.

[BWY79] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest-point problems. *Computer Science Department*, page 2451, 1979.

[Car97] G. F. Carey. Computational Grids: Generation, Adaptation and Solution Strategies. *Taylor & Francis*, June 1997.

[CB03] E. M. Cochrane and J. E. Beasley. The co-adaptive neural network approach to the Euclidean travelling salesman problem. *Neural Networks*, 16(10):1499–1525, 2003.

[CB08] 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"*, 2, 2008.

[CD97] P. Corke and P. Dunn. Real-time stereopsis using FPGAs. *IEEE TENCON—Speech and Image Technologies for Computing and Telecommunications*, pages 235–238, 1997.

[CDJN11] S. Chen, S. Davis, H. Jiang, and A. Novobilski. CUDA-based Genetic Algorithm on Traveling Salesman Problem. *Computer and Information Science*, pages 241–252, 2011.

[Ces96] G. Cesari. Divide and conquer strategies for parallel TSP heuristics. *Computer & Operations Research*, 23:681–694, July 1996.

[CGN+13] J. M. Cecilia, J. M. Garcia, A. Nisbet, M. Amos, and M. Ujaldon. Enhancing data parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Cimputing*, 73:42–51, January 2013.

[Cha05] S. Chambon. *Mise en correspondance stéréoscopique d'images couleur en présence d'occultations*. PhD thesis, Université Paul Sabatier, December 2005.

[CHKK12] J. C. Créput, A. Hajjam, A. Koukam, and O. Kuhn. Self-organizing maps in population based metaheuristic to the dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 24(4):437–458, 2012.

[Chr76] N. Christofides. The vehicle routing problem. *RAIRO-Operations Research-Recherche Opérationnelle*, 10(V1):55–70, 1976.

[CK06] J. C. Créput and A. Koukam. Local search study of honeycomb clustering problem for cellular planning. *International Journal of Mobile Network Design and Innovation*, 1(2):153–160, 2006.

[CK07a] J. C. Créput and A. Koukam. Interactive meshing for the design and optimization of bus transportation networks. *Journal of Transportation Engineering*, 133(9):529–538, 2007.

[CK07b] J. C. Créput and A. Koukam. Transport clustering and routing as a visual meshing process. *Journal of Information and optimization sciences*, 28(4):573–601, 2007.

[CK09] J. C. Créput and A. Koukam. A memetic neural network for the Euclidean traveling salesman problem. *Neurocomputing*, 72(4):1250–1264, 2009.

[CKH07] J. C. Créput, A. Koukam, and A. Hajjam. Self-organizing maps in evolutionary approach for the vehicle routing problem with time windows. *International Journal of Computer Science and Network Security*, 7(1):103–110, 2007.

[CKLC05] J. C. Créput, A. Koukam, T. Lissajoux, and A. Caminada. Automatic mesh generation for mobile network dimensioning using evolutionary approach. *Evolutionary Computation, IEEE Transactions on*, 9(1):18–30, 2005.

[CLK00] J. C. Créput, T. Lissajoux, and A. Koukam. A connexionnist approach to the hexagonal mesh generation. 2000.

[CTB06] I. Cabani, G. Toulminet, and A. Bensrhair. A fast and self-adaptive color vision matching:A first step for road obstacle detection. *Proceedings of IEEE Intelligent Vehicle Symposium*, pages 58–63, 2006.

[CW91] J. D. Crisman and J. A. Webb. The Warp machine on Navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:451–465, May 1991.

[DDGK13] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. *Journal of Parallel and Distributed Cimputing*, 73:52–61, January 2013.

[dGGV08] F. de Goes, S. Goldenstein, and L. Velho. A simple and flexible framework to adapt dynamic meshes. *Computer & Graphics*, pages 141–148, 2008.

[DGHW98] T. Darrell, G. Gordon, M. Harville, and J. Woodfill. Integrated Person Tracking Using Stereo, Color, and Pattern Detection. *Computer Vision and Pattern Recognition*, pages 601–608, 1998.

[DMMC09] S. Debattisti, N. Marlat, L. Mussi, and S. Cagnoni. Implementation of a simple genetic algorithm within the cuda architecture. *The Genetic and Evolutinary Computation Conference*, 2009.

[Dor92] M. Dorigo. *Optimization, learning and natural algorithm*. PhD thesis, Politecnico di Milano, 1992.

[dRAN07] R. L. M. E. do Rego, A. F .R. Araujo, and F. B. de Lima Neto. Growing Self-Organizing Maps for Surface Reconstruc tion from Unstructured Point Clouds . *Int. Joint Conf. Neural Netw.*, pages 1900–1905, 2007.

[FHZF93] O. Faugeras, B. Hotz, Z. Zhang, and P. Fua. Real time correlation-based stereo : Algorithm, implementation and applications. Technical Report Research Report RR-2013, August 1993.

[FKO+04] S. Forstmann, Y. Kanou, J. Ohya, S. Thuering, and A. Schmitt. Real-time stereo by using dynamic programming. *IEEE Conference on Computer Vision and Pattern Recognition*, page 29, June 2004.

[FRT97] A. Fusiello, V. Roberto, and E. Trucco. Efficient stereo with multiple windowing . *Proceedings of IEEE Conference on Computer Vision and Patern Recognition*, pages 858–863, June 1997.

[FRT00] A. Fusiello, V. Roberto, and E. Trucco. Sysmmetric stereo with multiple windowing. *International Journal of Pattern Recognition and Artificial Intelligence*, pages 1053–1066, 2000.

[FT11] N. Fujimoto and S. Tsutsui. A Highly-Parallel TSP Solver for a GPU Computing Platform. *Proceedings of the 7th international conference on Numerical methods and applications*, pages 264–271, 2011.

[Geo91] P.L. George. Génération automatique de maillages. . *Applications aux méthodes d'éléments finis,Masson, RMA 16*, 1991.

[GGK09] S. Grauer-Gray and C. Kambhamettu. Hierarchical belief propagation to reduce search space using CUDA for stereo and motion estimation. *Workshop on Applications of Computer Vision (WACV)*, pages 1–8, 2009.

[Gla13] P. Glaskowsky. Nvidia's Fermi: The first Complete GPU Computing Architecture. *http://www.nvidia.com/object/fermi-architecture.html*, 2013.

[GTX13] Nvidia Geforce GTX 570 review. Technical report, 2013.

[Gup12] N. Gupta. Thread and block heuristics in CUDA programming. 2012.

[GY05] M. Gong and Y. H. Yang. Near real-time reliable stereo matching using programmable graphics hardware. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1:924–931, June 2005.

[GY07] M. Gong and Y. H. Yang. Real-time stereo matching using orthogonal reliability-based dynamic programming. *IEEE Transactions on Image Processing*, pages 879–884, March 2007.

[HA97] J.F. Hamilton and J.E. Adams. Adaptive color plan interpolation in single sensor color electronic camera. *US patent 5,629,734, to Eastman Kodak Co., Patent and Trademark Office, Washington D.C.*, May 1997.

[Hal10] H. Halawana. *Partial demosaicing of CFA images for stereo matching*. PhD thesis, Université de Lille 1, 2010.

[HBG10] A. Hosni, M. Bleyer, and M. Gelautz. Near real-time stereo with adaptive support weight approaches. *Proc. 3DPVT*, 2010.

[HCK05] N. Hayari, J. C. Créput, and A. Koukam. A Neural Evolutionary Algorithm for Geometric Median Cycle Problem. *European Simulation and Modeling Conference*, (EUROSIS), 2005.

[Hel00] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[HIG02] H. Hirschmuller, P. R. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *International Journal of Computer Vision*, pages 229–246, June 2002.

[Hir05] H. Hirschmuller. Accurate and Efficient Stereo processing by Semi-Global matching and Mutual Information. *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and pattern Recognition*, 2:807–814, 2005.

[Hir08] H. Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE TPAMI*, pages 328–341, 2008.

[HK09] S. Hong and H. Kim. Memory-level and Thread-level Parallelism Aware GPU Architecture Performance Analytical Model. Master's thesis, ECE School of computer Science, 2009.

[HS06] Sarri Al Nashashibi Hiebert-Treuer B. and D. Scharstei. 2006 Stereo Datasets. *http:// vision.middlebury.edu/ stereo/ data/ scenes2006/* , Nov 2006.

[HS07] H. Hirschmuller and D. Scharstein. Evaluation of cost functions for stereo matching. *Proceedings of IEEE Conference on Computer Vision and Patern Recognition*, pages 1–8, June 2007.

[JD10] Nickolls J. and W. J. Dally. The gpu computing era. *IEEE Mocro*, 30:56–69, 2010.

[JJL08] A. Janiak, W. A. Janiak, and M. Lichtenstein. Tabu Search on GPU. *Journal of Universal Computer Science*, 14:2416–2427, 2008.

[JK90] C. S. Jeong and M. H. Kim. Parallel algorithm for traveling salesman problem on SIMD machines using simulated annealing. *Proceedings of the INternational Conference on Application Specific Array Processor*, pages 712–721, September 1990.

[JM97] D. S. Johnson and L. A. McGeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization, in: Local Search in Combinatorial Optimization*, pages 215–310. John WIley and Sons, 1997.

[JM02] D. S. Jognson and L. A. McGeoch. *Experimental analysis of heuristics for the STSP, in: Traveling salesman problem and its variations*, pages 369–443. Kluwer Academic publisher, 2002.

[Kar77] R. M. Karp. Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane. *Math. Ops Res.*, pages 209–224, 1977.

[KKK+95] S. Kimura, T. Kanade, H. Kano, A. Yoshida, E. Kawamura, and K. Oda. CMU Video-Rate Stereo Machine. *Mobile Mapping sysp.*, 1995.

[KL89] G. A. P. Kindervater and J. K. Lenstra. The Parallel Complexity of TSP Heuristics. *Journal of Algorithms*, 10, 249-270 1989.

[KO94] T. Kanade and M. Okutomi. A stereo matching algorithm with an adaptive window: Theory and experiment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 920–932, September 1994.

[Koh82] T. Kohonen. Clustering, Taxonomy, and Topological Maps of Patterns. *Proceedings of the 6th International Conference on Pattern Recognition*, October 1982.

[Koh01] T. Kohonen. Self-Organization Maps and associative memory. *Spring Verlag*, 2001.

[KON92] T. Kanade, M. Okutomi, and T. Nakahara. A multiple-Baseline Stereo method. *AppA Image Understanding Workshop*, pages 409–426, 1992.

[Kon97] K. Konolige. Small Vision Systems: Hardware and Implementation. *Eighth Int'l Sysmp. Robotics Research*, 1997.

[Kos93] A. Koschan. Dense stereo correspondence using polychromatic block matching. *International Conference on Computer Analysis of Images and Patterns*, 719 of Lecture Notes in Computer Science:538–542, September 1993.

[Kos96] A. Koschan. Using perceptual attributes to obtain dense depth maps. *IEEE Southwest Symposium on Image Analysis and Interpretation*, pages 155–159, April 1996.

[KSY+99] S. Kimura, T. Shinbo, H. Yamagushi, E. Kawamura, and K. Naka. A Convoler-based Real-time Stereo Machine (SAZAN). *Computer Vision and Pattern Recognition*, 1:547–463, 1999.

[KU03] D. Keysers and W. Unger. Elastic image matching is NP-complete. *Pattern Recognition Letters*, 24(1-3):445–453, 2003.

[KYO+96] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A stereo method for video-Rate Dense Depth Mapping and its New Applications. *Computer Vision and Pattern Recognition*, 1996.

[LLRG04] M. Labbé, G. Laporte, I. Rodriguez Martin, and J. J. S. González. The ring star problem: Polyhedral analysis and exact algorithm. *Networks*, 43(3):177–189, 2004.

[LLRG05] M. Labbé, G. Laporte, I. Rodriguez Martin, and J. J. S. González. Locating median cycles in networks. *European Journal of Operational Research*, 160(2):457–470, 2005.

[LS10] J. Liu and J. Sun. Parallel graph-cuts by adaptive bottom-up merging. *Proc. CVPR*, pages 2181 – 2188, 2010.

[Luo11] T. V. Luong. *Métaheuristiques parallèles sur GPU*. PhD thesis, Université Lille 1, December 2011.

[LWC03] 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*, pages 353–360, 2003.

[Mat92] L. Matthies. Stereo Vision for Planetary Rovers: Stochastic Modeling to Near Real-Time Implementation. *Journal of Computer Vision*, 8:71–91, 1992.

[Mat93] H. Matthies. A multi-DSP 96002 Board. Technical report, 1993.

[Mei11] Sun X., Zhou M., Jiao S., Wang H. and Zhang X. Mei X. On building an accurate stereo matching system on graphics hardware. *GPUCV*, 2011.

[MKLT95] L. Matthies, A. Kelly, T. Litwin, and G. Tharp. Obstacle Detection for Unmanned Ground Vehicles: A progress Report. *Intelligent Vehicles*, pages 66–71, 1995.

[MPL05] R. Y. Marc, M. Pollefeys, and S. Li. Improved real-time stereo on commodity graphics hardware. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1:36–36, 2005.

[MPP06] W. Miled, J. Pesquet, and M. Parent. Dense disparity estimation from stereo images. *Proceedings of 3rd International Sysposium on Image/Video Communications*, September 2006.

[MSH+12] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, and R. Hurley. Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA. volume 341, pages 012–018. IOP Publishing, 2012.

[Nec10] 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.

[Nis93] H. K. Nishihara. Real-time stereo- and Motion-based Figure-Ground Discrimination and Tracking Using LOG Sign-Correlation. *27th Asilomar Conf. Signals, Systems, and Computers*, pages 95–100, 1993.

[NVI10] NVIDIA. CUDA C best Practices Guide. 8 2010.

[NVI11] NVIDIA. CUDA Getting Started Guide (Linux). *http:// developer.nvidia.com/ nvidia-gpu-computing-documentation*, 2011.

[NVI12a] NVIDIA. CUDA C Programming Guide 4.2, CURAND Library, Profiler User's Guide. *http:// docs.nvidia.com/ cuda*, 2012.

[NVI12b] NVIDIA. NVIDIA CUDA CURAND Library. *http:// docs.nvidia.com/ cuda/ curand/ index.html*, 2012.

[NYYY07] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga. Implementation of an Effective Hybrid GA for Large-Scale Traveling Salesman Problems. *IEEE Transactions on Systems, Man, and Cybernetics*, Part B 37(1):92–99, 2007.

[OA05] Fermüller C. Ogale A. S. and Y. Aloimonos. Motion segmentation using occlusions. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 27(6):988–992*, June 2005.

[OHL+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96:879–899, 2008.

[Pap77] C. H. Papadimitriou. The Euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

[PCA98] G. Pajares, J. M. Cruz, and J. Aranda. Stereo matching based on the self-organizing feature-mapping algorithm. *Pattern Recognition Letters*, 19(3):319–330, 1998.

[PJ09] P. Pospochal and J. Jaros. GPU-based Acceleration of the Genetic Algorithm. July 2009.

[PMM88] J. F. Pekny, D. L. Miller, and G. J. McRae. Application of a parallel traveling salesman problem algorithm to no-wait flowshop scheduling. Technical report, Department of Chemical Engineering, 1988.

[PSB+07] 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*, pages 198–210, 2007.

[RBL04] J. Renaud, F. F. Boctor, and G. Laporte. Efficient heuristics for median cycle problems. *Journal of the Operational Research Society*, 55(2):179–186, 2004.

[Rei91] G. Reinelt. TSPLIB—A Traveling Salesman Problem Library. *ORSA journal on computing*, 3(4):376–384, 1991.

[RHB+11] C. Rhemann, A. Hosni, M. Bleyer, C. Rother, and M. Gelautz. Fast cost-volume filtering for visual correspondence and beyond. *Proc. CVPR*, 2011.

[SB08] Luis Moura E Silva and Rajkumar Buyya. *Parallel Programming Models and Paradigms*. PhD thesis, Universidade de Coimbra, 2008.

[Sch00] R. Schneiders. Algorithms for Quadrilateral and Hexahedral Mesh Generation. *Proceedings of the VKI Lecture series on Computational Fluid Dynamic*, pages 2000–2004, 2000.

[SK10] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[Sol87] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[SOS97] L. Shi, S. Olafsson, and N. Sun. New Parallel Randomized Algorithms for the Traveling Salesman Problem. Technical report, Department of Industrial Engineering, March 1997.

[SRS+08] Ryoo S., C. I. Rodrigues, S. S. Stone, J. A. Stratton, S. Z. Ueng, S. S. Baghsorkhi, and Hwu W. W. Program optimization carving for gpu computing. *J. Parallel Distributed Computing*, 68:1389–1401, 2008.

[SS02] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two- frame stereo correspondence algorithms. *International Journal of Computer Vision*, pages 7–42, 2002.

[SST91] O. Sarzeaud, Yann Stephan, and Claude Touzet. Finite Element Meshing using Kohonen's Self-Organizing Maps. *International Conference on Artificial Neural Network*, juin 1991.

[Stü98] T. Stützle. Parallelization Strategies for Ant Colony Optimization. pages 722–731. Springer-Verlag, 1998.

[SWP+12] C. Shi, G. Wang, X. Pei, H. Bei, and X. Lin. High-accuracy stereo matching based on adaptive ground control points. *IEEE TIP*, 2012.

[TRFR98] E. G. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel Ant Colonies for Combinatorial Optimization Problems. pages 239–247, 1998.

[Vek99] O. Veksler. *Efficient Graph-based Energy Minimization Methods in Computer Vision*. PhD thesis, Cornell University, 1999.

[Vek05] O. Veksler. Stereo correspondence by Dynamic programming on a tree. *Computer Vision and Pattern Recognition*, pages 384–390, June 2005.

[VGB09] M. Vanetti, I. Gallo, and E. Binaghi. Dense two-frame stereo correspondence by self-organizing neural network. In *Image Analysis and Processing–ICIAP 2009*, pages 1035–1042. Springer, 2009.

[WDZ09] J. Wang, J. Dong, and C. Zhang. Implementation of Ant Colony Algorithm based on GPU. *Sixth International Conference on Computer Graphics, Images and VIsualization*, 2009.

[Web93] J. A. Webb. Implementation and performance of Fast parallel multi-baseline Stereo Vision. *DARPA Image Understanding Workshop*, pages 1005–1012, 1993.

[WGY06] L. Wang, M. Gong, and R. Yang. How far can we go with local optimization in real-time stereo matching. *In proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 129–136, June 2006.

[WLG+06] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. *3D Data Processing Visualization and Transmission - 3DPVT*, pages 798–805, 2006.

[WmS+05] L. Wang, A. A. maciejewski, H. J. Siegel, V. P. Roychowdhury, and B. D. Eldridge. A STUDY OF FIVE PARALLEL APPROACHES TO A GENETIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM. *Intelligent Automation and Soft Computing*, 11:217–234, 2005.

[Wor07] J. A. Worby. Multi-resolution graph cuts for stereo-motion estimation. Master's thesis, University of Toronto, 2007.

[WTFJ10] Y. Wei, C. Tsuhan, F. Franz, and C. H. James. High performance stereo vision designed for massively data parallel platforms. *IEEE TCSVT*, pages 1–11, 2010.

[Wu00] J. Wu. Bayesian estimation of stereo disparity from phase based measurements . Master's thesis, Queen's University, February 2000.

[WV97] J. Woodfill and B. Von Herzen. Real-time stereo Vision on the PARTS Reconfigurable Computer. *IEEE Workshop FPGAs for Custom Computing machines*, pages 242–250, 1997.

[WZC13] H. Wang, N. Zhang, and J. C. Créput. A Massive Parallel Cellular GPU implementation of Neural Network to Large Scale Euclidean TSP. *12th Mexican International Conference on Artificial Intelligence, MICAI 2013*, November 2013.

[YCP05] Q. Yu, C. Chen, and Z. Pan. Parallel genetic Algorithms on Programmable Graphics Hardware. *Lecture Notes in Compter Science 3612*, page 1051, 2005.

[YEA08] Q. Yang, C. Engels, and A. Akbarzadeh. Near real-time stereo for weakly-textured scenes. *Proc. BMVC*, pages 80–87, 2008.

[YIL08] M. Yoon, I. P. Ivrissimtzis, and S. Lee. Self-organising maps for implicit surface reconstruction. In *TPCG*, pages 83–90. Citeseer, 2008.

[YKN+12] 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*, 5(3), 2012.

[YLD07] Y. Yang, O. Losson, and L. Duvieubourg. Quality evaluation of color demosaicing according to image resolution. *Proceedings of the 3rd International Conference on Signal-Image Technology & Internet-based Systems, pages 640–646,Shanghai Jiaotong University, China*, December 2007.

[You09] Y. S. You. Parallel Ant System for Traveling Salesman Problem on GPUs. *GECCO 2009—GPUs for Genetic and Evolutionary Computation*, pages 1–2, 2009.

[YWY+09] Q. Yang, L. Wang, R. Yang, H. Stewenius, and D. Nister. Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling. *IEEE TPAMI*, pages 492–504, 2009.

[ZB06] Y. Zhang and C. Bajaj. Adaptive and quality quadrilateral/hexahedral meshing from volumetric data. *Comput. Methods Appl. Mech. Eng.,*, pages 942–960, 2006.

[ZCW+13] N. Zhang, J. C. Créput, H. Wang, C. Meurie, and Y. Ruichek. Partial demosaicing for stereo matching of CFA images on GPU and CPU. *The Third International Conference on Advanced Communications and Computation, INFOCOMP 2013*, November 2013.

[ZLL09] K. Zhang, J. Lu, and G. Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE TCSVT*, pages 1073–1079, 2009.

[ZW94] R. Zabih and J. Woodfill. Non-parametric Local Transforms for Computing Visual Correspondence. *Third European Conf. Computer Vision*, pages 150–158, 1994.

[ZWC+13a] N. Zhang, H. Wang, J. C. Créput, J. Moreau, and Y. Ruichek. A Near Real-Time Color Stereo Matching Method for GPU. *Proceedings in The Third International Conference on Advanced Communications and Computation, INFOCOMP 2013*, 6, 2013.

[ZWC+13b] N. Zhang, H. Wang, J. C. Créput, J. Moreau, and Y. Ruichek. Cellular GPU Model for Structured Mesh Generation and its Application to the Stereo-Matching Disparity Map. *The IEEE International Symposium on Multimedia, ISM 2013*, December 2013.

Abstract:

The work presented in this PhD studies and proposes cellular computation parallel models able to address different types of NP-hard optimization problems defined in the Euclidean space, and their implementation on the Graphics Processing Unit (GPU) platform. The goal is to allow both dealing with large size problems and provide substantial acceleration factors by massive parallelism. The field of applications concerns vehicle embedded systems for stereovision as well as transportation problems in the plane, as vehicle routing problems. The main characteristic of the cellular model is that it decomposes the plane into an appropriate number of cellular units, each responsible of a constant part of the input data, and such that each cell corresponds to a single processing unit. Hence, the number of processing units and required memory are with linear increasing relationship to the optimization problem size, which makes the model be able to deal with very large size problems.
The effectiveness of the proposed cellular models has been tested on the GPU parallel platform on four applications. The first application is a stereo-matching problem. It concerns color stereovision. The problem input is a stereo image pair, and the output a disparity map that represents depths in the 3D scene. The goal is to implement and compare GPU/CPU winner-takes-all local dense stereo-matching methods dealing with CFA (color filter array) image pairs. The second application focuses on the possible GPU improvements able to reach near real-time stereo-matching computation. The third and fourth applications deal with a cellular GPU implementation of the self-organizing map neural network in the plane. The third application concerns structured mesh generation according to the disparity map to allow 3D surface compressed representation. Then, the fourth application is to address large size Euclidean traveling salesman problems (TSP) with up to 33708 cities.

Résumé :

Le travail présenté dans ce mémoire étudie et propose des modèles de calcul parallèles de type cellulaire pour traiter différents problèmes d'optimisation NP-durs définis dans l'espace euclidien, et leur implantation sur des processeurs graphiques multi-fonction (Graphics Processing Unit; GPU). Le but est de pouvoir traiter des problèmes de grande taille tout en permettant des facteurs d'accélération substantiels à l'aide du parallélisme massif. Les champs d'application visés concernent les systèmes embarqués pour la stéréovision de même que les problèmes de transports définis dans le plan, tels que les problèmes de tournées de véhicules. La principale caractéristique du modèle cellulaire est qu'il est fondé sur une décomposition du plan en un nombre approprié de cellules, chacune comportant une part constante de la donnée, et chacune correspondant à une unité de calcul (processus). Ainsi, le nombre de processus parallèles et la taille mémoire nécessaire sont en relation linéaire avec la taille du problème d'optimisation, ce qui permet de traiter des instances de très grandes tailles.
L'efficacité des modèles cellulaires proposés a été testée sur plateforme parallèle GPU sur quatre applications. La première application est un problème d'appariement d'images stéréo. Elle concerne la stéréovision couleur. L'entrée du problème est une paire d'images stéréo, et la sortie une carte de disparités représentant les profondeurs dans la scène 3D. Le but est de comparer des méthodes d'appariement local selon l'approche winner-takes-all et appliquées à des paires d'images CFA (color filter array). La deuxième application concerne la recherche d'améliorations de l'implantation GPU permettant de réaliser un calcul quasi temps-réel de l'appariement. Les troisième et quatrième applications ont trait à l'implantation cellulaire GPU des réseaux neuronaux de type carte auto-organisatrice dans le plan. La troisième application concerne la génération de maillages structurés appliquée aux cartes de disparité afin de produire des représentations compressées des surfaces 3D. Enfin, la quatrième application concerne le traitement d'instances de grandes tailles du problème du voyageur de commerce euclidien comportant jusqu'à 33708 villes.

SPIM

■ École doctorale SPIM - Université de Technologie Belfort-Montbéliard
F - 90010 Belfort Cedex ■ tél. +33 (0)3 84 58 31 39
■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

utbm
université de technologie
Belfort-Montbéliard