Graphics Processing Units for Constraint Satisfaction

Malek Mouhouband and Ahmed Mobaraki
Dept. of Computer Science, Univ. of Regina, Regina, Canada

Keywords: Constraint Satisfaction, Graphics Processing Units, Parallel Computing.

Abstract: A Constraint Satisfaction Problem (CSP) is a powerful formalism to represent constrained problems. A CSP includes a set of variables where each is defined over a set of possible values, and a set of relations restricting the values that the variables can simultaneously take. There are numerous problems that can be represented as CSPs. Solving CSPs is known to be quite challenging in general. The literature poses a great body of work geared towards finding efficient techniques to solve CSPs. These techniques are usually implemented in a system commonly referred to as a constraint solver. While many enhancements have been achieved over earlier ones, solvers still require powerful resources and techniques to solve a given problem in a reasonable running time. In this paper, a new parallel-based approach is proposed for solving CSPs. In particular, we design a new CSP solver that exploits the power of graphics processing units (GPU), which exist in modern day computers, as an affordable parallel computing architecture.

1 INTRODUCTION

A Constraint Satisfaction Problem (CSP) is a mathematical formalism used for representing a variety of constrained problems ranging from scheduling to bioinformatics (Dechter, 2003). More formally, a CSP includes a set of variables, each defined on a domain of finite and discrete values, and a set of constraints restricting the values tha the variables can simultaneously take. A solution to a CSP is a set of assignments of values to variables such that all constraints are satisfied. A CSP is known to be a NP-hard problem and requires a solving algorithm of exponential time cost. Therefore, given a CSP, one usually needs advanced solvers and techniques to find one or more feasible solutions (Mackworth, 1977; Haralick and Elliott, 1980; Dechter, 2003; Bessière et al., 2005; Lecoutre and Tabary, 2008; Balafoutis and Stergiou, 2010; Mouhoub and Jashmi, 2011; Abbasian and Mouhoub, 2016). There are several systems that are developed specifically for solving CSPs (Lecoutre and Tabary, 2008). Such systems have been developed in many different manners to meet the requirements of different CSP problems. Still, most of the systems suffer some limitations with regards to how fast they can find a solution. Most solvers are dedicated to run on personal computers. This means they run in sequential order until a solution is found, if it exists. This becomes more challenging when the problem has a very large number of variables with a large domain size. Even though there are several improvements that have been proposed to enhance the performance of CSP solvers (Bessière et al., 2005; Lecoutre and Tabary, 2008; Balafoutis and Stergiou, 2010; Mouhoub and Jashmi, 2011; Abbasian and Mouhoub, 2016), there are still some difficulties with hard to solve problems. One well-known improvement is to build a parallel CSP solver that uses multiple processors. Such an idea performs very well as long as the user has the ability to acquire powerful machines. Besides, such machines often come with unaffordable cost. Another suggested idea is to implement a parallel CSP solver that takes advantage of the multi-core processor of the personal computer platform. This approach seems very promising, as it only requires a personal computer. However, the system developer would need to maintain balance within the multi-core processor usability. That is to say, while an application is using multiple cores, this application would be required to assign tasks to each of the cores in such way all of them are used equally.

In this paper, we propose a new GPU-based system for solving CSPs. This is achieved by adopting the CUDA framework (Cook, 2012). CUDA is a programming platform that was developed by NVIDA to allow programmers to use a NVIDIA Graphics Processing Unit (GPU). As a programmable and general-purpose architecture, GPU can be purchased as cheap as fifty dollars. Moreover, a GPU outperforms the Central Processing Unit (CPU), in various regards.

Mouhoub M. and Mobaraki A.
Graphics Processing Units for Constraint Satisfaction.
DOI: 10.5220/0006214806530657
In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, pages 653-657
ISBN: 978-989-758-220-2
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

For example, while the CPU often involves up to 16 cores, GPU can have thousands cores due to the fact that GPU has been developed to work as a parallel computing architecture, mainly for image rendering and other graphics and media computations. Another advantage of using GPU as a parallel platform is that NVIDIA has introduced a new programming language called CUDA C, which is an extended version of C language, to help developers monopolize all the resources of the GPU without having to learn a new programming language. The CUDA platform will be responsible for scheduling the threads and avoiding any misbalancing which could lead to starvation.

The rest of the paper is organized as follows. First, Section 2 offers background information about CUDA, along with some details for the advantages of using this new computing platform. Then, a description of the proposed system for solving CSPs with a GPU is presented in Section 3. Finally, concluding remarks and future works are reported in Section 4.

2 COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

The GPU has been designed to relieve the CPU from the highly computation processes. The GPU has been responsible for graphics processing like image rendering. This is why it comes with a large number of transistors which are mostly dedicated for data processing, in comparison to the CPU which requires more processing control and caching the memory. Moreover, GPU memory has a higher bandwidth than CPU which speeds up the performance of the arithmetic operations on the data. The reason for this is because the main task of the GPUs is to execute the same operations on a larger number of data, which is what the image rendering is about. However, CPUs are general architecture designed to execute all types of functions. In the image rendering processing, or any other graphics processing, the main idea is to divide the problem data into smaller blocks in which a block can perform the same work on the sub-problem. Thus, any problem that processes a repeatedly arithmetic operation on a larger number of data can be partitioned into a smaller size and tackled with as independent from other partitions of the problem as long as the partitioned data elements are independent from each other. In 2006, NVIDIA announced a new parallel computing architecture called Compute Unified Device Architecture, or CUDA (Cook, 2012). It exploits the parallelism of the GPU hardware to its highly computation for solving highly computational problem out the field of graphics. In contradiction with CPU, CUDA architecture would perform parallel computing work in a more professional way than CPU. Also, NVIDA has developed a software programming environment that allows the developers to use it for implementing their applications with a CUDA C that is an extension programming language of the C Language. One of the obstacles of a multicore processor is that developers need to develop an application that not just uses the multicore technology in the CPU, but also maximizes the benefits of the parallel computation to its high level to reach the top speedup that can be achieved by the multicore processor. Still, this requires a lot of programming experience and understanding of the CPU architecture. These difficulties have been tackled with CUDA by introducing the new programming model with instructions sets based on the C programming language. Developers who are not familiar with the CUDA platform can still perform well in programming with CUDA C and reach a very high leverage of the GPU parallel computing ability. A developer needs only to write the kernel that will run on the GPU, and the CUDA platform will execute this kernel on all of the elements of the data. CUDA C also allows the users to write a heterogeneous code that runs on the CPU and GPU in the same file. The code that concerns the CPU called host code while the part that runs on the GPU is called the device code. In CUDA C, there are a few new instructions that are added to the extension of C programming language. ThreadIdx is the keyword for accessing the current thread ID. It is used to direct the specific thread work.

In Figure 1, a sample code, for example, shows how to add two vectors and store the results in a third one. The developer only needs to write two functions; one that will be executed on the CPU and will be responsible for copying the data from the from CPU memory to GPU memory, triggering the kernel, synchronizing, then copying back the result form device memory to the CPU memory. This code is written in the main function and it is executed only once. While in the Kernel section, the code will run equal to the number of threads that are already specified by the host code in the many function when the kernel is triggered. In this example, the thread number is equal to the number of array elements. Each thread will access different elements in the array of A and adds it to the correspondent element in array B, then store the result in the correspondent element in array C. After the kernel function is complete, the main function of the host will be to direct the CPU to copy back the elements of array C from the device memory to the CPU memory, and the program will terminate.

```
// Kernel definition
__global__ void VecAdd(float* A,
                       float* B,
                       float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figure 1: Vectors Addition in CUDA C language.

3 PROPOSED SYSTEM

3.1 System Overview

Our proposed system is composed of four major components that are running on the same local machine or platform. The components are Problem, XCSP Parser (Lecoutre and Tabary, 2008), Consistency Algorithm (Dechter, 2003; Bessière et al., 2005), and finally GPU Solver. Each one of them, except Problem, performs a different task. The system is discussed in more details in the next subsection. The main component of the whole system is the Problem class. This latter is provided with the basic functionalities that are required to create, manage, and find the solution for a CSP. These functionalities are: create a CSP, add and remove variables, add and remove constraints and find solutions. It is also provided with the ability to retrieve a CSP data that is stored in an XML file, parse them, and feed that data to the problem class. In addition, the Problem class can also run a local consistency algorithm on a problem in order to trim the CSP variables domain before submitting it to the GPU Solver component. The Consistency component is composed of two types of consistency techniques: arc (Dechter, 2003; Bessière et al., 2005) and path consistency (Bessière, 1996) algorithms. Both of them are concrete classes that implement the interface class called Consistency. Consistency interface defines the basic attributes and methods of local consistency algorithms. The most important here, the interface declares a function called run in which the target algorithm will be triggered over the CSP. Therefore, both of the implementations, or any other future classes, must implement this function. At the concrete classes, arc and path, the corresponding technique is implemented. Thus, when the consistency function of the problem class is invoked, one of these implementations is triggered based on the choice given by the user. The GPU Solver is a new implementation that is proposed to improve the CSP solver system by using CUDA architecture for parallel computing. Using the Cudafy library, we can create several system running threads that match the number of all different locally consistent variable assignments that can be generated by a given CSP problem. Each thread will find the corresponding assignment for the variables based on its identification number that is assigned to each thread by the GPU. Certainly, each GPU has a maximum number of threads that can be used. Those variables considered in the process of generating the combinations are called partitioned variables. Consequently, the unselected variables are called unpartitioned variables, and are assigned to a value from there domains sequentially using the backtracking algorithm following a given constraint propagation strategy (Haralick and Elliott, 1980). The GPU Solver class is composed of two main functions. The first is FLABT, which implements the backtracking algorithm with Full Look Ahead as a propagation strategy. The other function is FCBT, which implements the backtracking algorithm with the Forward checking strategy. Either of them is to be invoked from function problem. FindSolution and the selection will be based on the user choice that is provided as a parameter for this function. Based on the CSP data, the GPU Solver will instantiate an adequate number of threads that is equal to number of all the combinations of the given CSP as long as it does not exceed the maximal number threads that can be created by the GPU. In case a CSP combination number is larger than the maximal number of threads, some of these variables' domains will be partitioned while the rest will be solved using the backtracking algorithm. Once the required number of threads is specified, the CSP problem data will be copied from the CPU memory to the GPU memory, and then the correspondence kernel will be invoked to run on the GPU. The system needs to synchronize now until all the executions of the GPU are performed. Once this step is complete, the system retrieves the results from the GPU memory and frees all the resource allocations being used by the GPU, and the next step will be storing the results in the problem class data structure, namely solutions. However, this is how the GPU Solver performs on finding the solution, but not how it is done on the GPU kernel. For the FLABT kernel, the data required to solve a CSP is domains, constraints, vari-

ables indexes, in additions to the number of threads. There are as many versions of all the variables' domains as the number of threads so that each of the threads can access a separate version of all the variables' domains and update them separately. The first step is that each thread finds a unique combination based on its ID number that is assigned by the GPU. If all the combinations of the given CSP can be generated by the threads, each thread will invoke a function called a propagation function for testing the legibility of this solution using the constraints table of the CSP. Thus, the kernel function will terminate itself at that level. On the other hand, if the number of threads is less than the entire combinations, the selected variables will be assigned via each of the threads, those assignments are tested through the propagation function. Only those threads that carry legal combinations are considered, and all other threads will be terminated. Afterward, the backtracking algorithm combined with a constraint propagation technique, either forward checking or Full Look ahead, will be executed on the rest of threads. The third step after the completion of backtracking algorithm is to examine all the threads against the constraints table to find combinations that satisfy all the constraints.

3.2 Software Architecture and Description

The layout of the system (see Figure 2) consists of two interfaces, the core system and the hardware. The user can enter a CSP data via graphics user interface or by submitting XML file constraints that includes all the information needed to create a problem. XML Parser is responsible for parsing the XML file and extracting the data of the CSP and feeding it to the core system. The problem data has to be written in a certain format that is specified in (Lecoutre and Tabary, 2008). Once a problem has been created, the core system is ready to be initiated and perform its functionalities over the problem. The system use cases diagram is depicted in Figure 3.

As described before, our proposed software implements four main classes: Problem, XCSP parser, consistency, and GPU Solver. Their tasks are to represent a CSP problem, reading a problem data from a file, performing consistency algorithm, and finding the solution, respectively. The main class is the problem class, in which all other classes' functionalities are accessed. Figure 4 describes all the classes and the integrations between them.

There are two main functionalities for the systems, removing inconsistent values and finding the solution of the problem. The Problem Class implements each

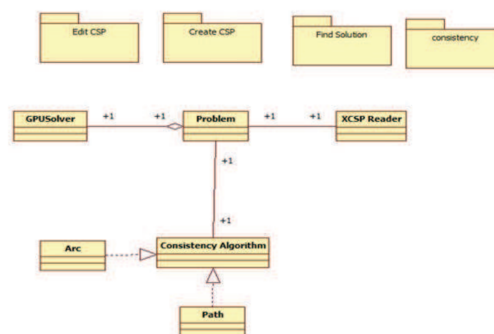

Figure 2: System interaction component.

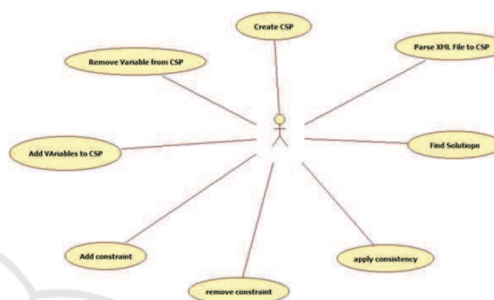

Figure 3: Use Case Diagram.

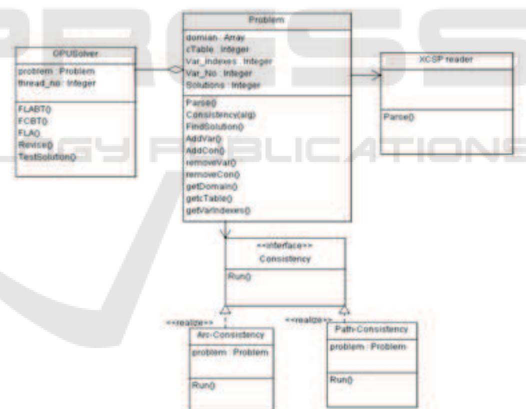

Figure 4: Class diagram.

of these functionalities in a function that can be called from the main class. FindSolution functionality is depicted in Figure 4. The function requires a string parameter and has no value to return since the results will be stored in an array of two dimensions, namely solutions. FindSolution will invoke the proper algorithm function of the GPUSolver class depending on the parameter being passed to the function. There are two implementations of the backtracking algorithms, FCBT and FLABT. Each one of the backtracking algorithm functions invoke in turn a different kernel to run on GPU cores. More specifically, FCBT will invoke FindWithFC kerenel, while FLABT will invoke

the FindwithFLA kernel. The FindSolution function will pass its class, problem, as a parameter to the backtracking function in GPUSolver class, and the backtracking function in turn will copy the problem data from CPU memory to the GPU memory. Right now, everything is ready for the kernel to be invoked. The backtracking function, either FCBT or FLABT, needs to synchronize until the kernel finishes its work. The kernel will store all the solutions that are parallelized in a two-dimension array, namely Solutions. This is the array that will be retrieved from the GPU memory by the FLADBT function after the CPU has been synchronized in order for the kernel to finish its task. Of course, not all the solutions found are legitimate. That is why there is another array called Flag, which holds a flag value for each corresponding solution.

Our system is an open source programming environment produced by NVIDA that allows a developer to implement a kernel and execute it on the graphics processing unit. The programming language used is an extended C language. Cudafy is a library developed by C# to allow developers to develop their applications in a high level programming language like C# and they do not need prior knowledge about the C language structure or the parallel computing architect. Still, the application needed to have a NVCC compiler to compile and execute the output of the Cuadfy code. This library target .Net is designed to be used in the Visual Studio.Net Framework, particularly for C# programmers, so that the developed application can be executed, partially not entirely, in a parallel manner.

4 CONCLUSION AND FUTURE WORK

In this work, we proposed a GPU-based solver for CSPs. Most of the solvers existing today rely on a CPU processing to solve a CSP problem. After enforcing constraint propagation, the CSP can be composed of locally consistent combinations such that some of them form a consistent solution. GPU can be programmed to distribute multiple threads over these combinations and find which of those combinations would satisfy the constraints. The GPU is easily programmable and does not require any advance knowledge for thread scheduling, maintaining, or synchronizing. Finally, the GPU can outperform classic solving methods by partitioning the CSPs and parallelizing the threads. This often leads to large savings in terms of computations. The following will be considered in the near future: developing a Graphics User Interface, implementing more constraint techniques including variable ordering heuristics (Balafoutis and Stergiou, 2010; Mouhoub and Jashmi, 2011), considering other forms of paralllel architectures for solving CSPs (Abbasian and Mouhoub, 2013; Abbasian and Mouhoub, 2016), and generalizing the system to adopt other types of NVIDIA GPU.

REFERENCES

Abbasian, R. and Mouhoub, M. (2013). A hierarchical parallel genetic approach for the graph coloring problem. *Applied Intelligence*, 39(3):510–528.

Abbasian, R. and Mouhoub, M. (2016). A new parallel ga-based method for constraint satisfaction problems. *International Journal of Computational Intelligence and Applications*, 15(03).

Balafoutis, T. and Stergiou, K. (2010). Conflict directed variable selection strategies for constraint satisfaction problems. In *Hellenic Conference on Artificial Intelligence*, pages 29–38. Springer.

Bessière, C. (1996). A simple way to improve path consistency processing in interval algebra networks. In *AAAI'96*, pages 375–380, Portland.

Bessière, C., Régin, J., Yap, R. H. C., and Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2):165–185.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Haralick, R. and Elliott, G. (1980). Increasing tree search efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313.

Lecoutre, C. and Tabary, S. (2008). Abscon 109: a generic csp solver. In *2nd International Constraint Solver Competition, held with CP'06 (CSC'06)*, pages 55–63.

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

Mouhoub, M. and Jashmi, B. J. (2011). Heuristic techniques for variable and value ordering in csps. In Krasnogor, N. and Lanzi, P. L., editors, *GECCO*, pages 457–464. ACM.