2006 IEEE Congress on Evolutionary Computation
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Cellular Genetic Algorithms and Local Search for 3-SAT problem on Graphic Hardware

Zhongwen Luo and Hongzhi Liu

Abstract—**As a well known NP-hard problem, SAT problem is widely discussed by computer science society. In this paper, two common algorithms for SAT problems are implemented based on graphic hardware. They are greedy local search and genetic algorithm. After a brief description of the basic algorithm, we give our modification of the algorithm for fitting with graphic hardware's SIMD model. Then some implementation details and tricks for graphic hardware are given. At last testing result are given. We compared the performance of graphic hardware based computation with CPU based computation. The result shows a performance improvement on graphic hardware over ordinary CPU.**

I. Introduction

As an NP-complete problem [1], the satisfiability (SAT) problem is widely researched by the computer science society. Many practical problems, such as graph coloring, job-shop scheduling, and real-world scheduling, can be represented as SAT problem.

As for the computation of this problem, there are two main classes of algorithms. One is the greedy local search [2], the other is genetic algorithms [3][4]. One of the most popular greedy local search procedure methods is called GSAT [2]. A more efficient algorithm is the random walk SAT (WSAT) [5], which is an extension of GSAT that mixed a random walk strategy with greedy local search.

Because of the problem's complexity, for large size problem, it usually takes a long time to solve the SAT using genetic algorithm. Parallel computers are often used for solving this kind of problem. Based on a parallel machine Meiko CS-2, Gianluigi Folino et al. used the parallel cellular genetic algorithm (CGA) [2][6][7] to solve this problem.

Motivated by the internal parallelism, powerful computing ability and the fast growth of performance, graphic process unit(GPU) is used for solving SAT problems. Some simple genetic algorithm on GPU for 3-SAT has been implemented by Z. Luo[8] and Thompson, C. J[9], but both algorithms are not fully complete and their target are to show the performance improvement of GPU over CPU. In this paper, we will implement a fully complete cellular genetic algorithm on GPU.

The random k-SAT problems are a family of SAT problems obtained by randomly generating instances with respect to three parameters: the number of variables, the number of clauses and the length(k) of each clause . Mitchell et al. [10] have shown that with k =3 these problems are very hard when the generated instances are equally likely to be either satisfiable or unsatisfiable. So we mainly discuss 3-SAT problems.

II. Greedy Search Algorithm for 3-SAT on GPU

A. GSAT

One of the most popular methods for solving SAT is the greedy local search method for SAT (GSAT), which starts with a randomly generated truth assignment. It then changes (flips) the assignment of the variable that leads to the largest decrease in the total number of unsatisfied clauses. Such flip is repeated until either a satisfying assignment is found or a preset of maximum number of flips is reached. This process is repeated as needed up to a maximum of *Maxtries* times. The basic GSAT algorithm is shown below.

```
  Algorithm: Greedy Local Search for 3-SAT
problem
  Input: a set of clauses CL, and integers
Maxflips and Maxtries.
  Output: a satisfying truth assignment of CL,
if any is found.
  for (i = 1; i< Maxtries ; i++)
  {
    T=generate();//    a    randomly    truth
assignment;
    for (j = 1; j< Maxflips ; j++)
    {
    if T satisfies CL then return T;
    for (p=0;p<vari_no;p++)
    {
    // No. of clauses unsatisfied by T and
    // satisfied by reverse (flip) p.
    Make = makep(p);
    Break = breakp(p);
    Diff[p] = Make - Break;
    }
    maxDiffList = list of variables with the
greatest Diff;
    p = a random member of MaxDiffList;
    T = T with the truth assignment of p flipped;
    }
  }
  return "no satisfying assignment found"
```

This work was supported in part by the Scientific Research Foundation for the Returned Overseas Chinese Scholars, China University of Geosciences, Wuhan.

Zhongwen Luo is with the Faculty of Information Engineering, China University of Geoscience, Wuhan, 430074 China. He a also a PhD candidate with the China University of Geoscience(Beijing), Beijing, 100080, China, (86-27-67883723; fax: 86-27-67883728; e-mail: luozw@cug.edu.cn).

Hongzhi Liu was with Faculty of Information Engineering, China University of Geoscience, Wuhan, 430074 China. He is now with the School of Software and Microelectronics, Beijing University, Beijing, 100080 China (e-mail: liuhz87@yahoo.com.cn).

0-7803-9487-9/06/$20.00/©2006 IEEE

B. Random Walk for GSAT

The main problem in applying local search methods to SAT problems is that the search space presents a lot of local optima and consequently the algorithm is trapped at local minima. In order to overcome this problem, one method is allowing moving to a new neighbor point of the local space even if the value of the object function decreases. One of these methods is referred to as WSAT[5], which is an extension of GSAT.

1) Random Walk Strategy I: The random walk strategy introduced in [5] is shown as following:

With probability p, pick a variable occurring in some unsatisfied clause and flip its truth assignment.

With probability 1- p, follow the standard GSAT scheme, i.e., pick randomly form the list of variables that give the largest decrease in the total number of unsatisfied clauses.

2) Random Walk Strategy II: Randowm Walk strategy I is not suited for the implementation on GPU, because each case in the strategy needs multiple passes to complete. According to a randomly generated number *prand,* different cell in CGA may be in different case, and they may need different operation in the same time. This is impossible for the GPU's SIMD architechture. So we propose a simple random walk strategy for graphic hardware realization.

If maxDiff >0, then follow the standard GSAT scheme, i.e., pick randomly form the list of variables that give the largest decrease in the total number of unsatisfied clauses.

Else random choose some variables and flip its truth assignment.

3) Random Walk Strategy III: Strategy II is simple and suited for GPU, but performance is poor. So a new and more efficient random walk strategy is needed. We device a new strategy as follow:

If maxDiff >0, then follow the standard GSAT scheme, i.e., pick randomly form the list of variables that give the largest decrease in the total number of unsatisfied clauses.

Else random choose some variables occurring in part of the unsatisfied clause and flip its truth assignment.

This strategy is similar to the standard random walk strategy, and also suitable for GPU programming.

III. Cellular Genetic Algorithm for 3-SAT on GPU

Genetic Algorithms (GAs) are a family of computational models inspired by evolution [12].

A. The Canonical Genetic Algorithms

A canonical genetic algorithm evolves a constant size population of individuals (called chromosomes) by using the genetic operator of reproduction, recombination and mutation.

The canonical Genetic Algorithms work as following:

1. Initialization: randomly generated a population of *n l*-bit chromosomes (candidate solutions to a problem).

2. Evaluation: Calculate the fitness $f(x)$ of each chromosome x in the population.

3. Repeat until the termination condition satisfied

.a. Selection: select a pair of parent chromosomes from the current population according to the fitness.

.b. Recombination: with probability p_c, cross over the pair at a randomly chosen point to form two offspring.

.c. Mutation: with probability p_m, mutate the two offspring.

d. Evaluation: Calculate the fitness $f(x)$ of each chromosome x in the population.

The SAT problem can be directly mapped onto a population of chromosomes. Each chromosome is a string of m bits which can be interpreting as an assignment of truth value to the set of m variables of SAT problem. The i-th bit represents the truth value of the i-th Boolean variable: true if the bit value is 1, false otherwise. The fitness of a chromosome takes as the number of unsatisfied clauses, and a string whose fitness is zero is the solution to the SAT problem.

B. the Cellular Genetic Algorithms

Cellular genetic algorithm (CGA) is a kind of fine grained parallel implementation of genetic algorithm. In CGAs, individuals are placed on a toroidal d-dimensional grid, each individual located in a grid (referred as a cell). Every individual has a neighbor, and each individual can only mated with other individuals in its neighborhood.

For CGA, a genetic algorithm works on each cell independently. The only difference between the genetic algorithm used in CGA and the canonical genetic algorithm is the selection operator. In CGA, parents are selected from neighborhood.

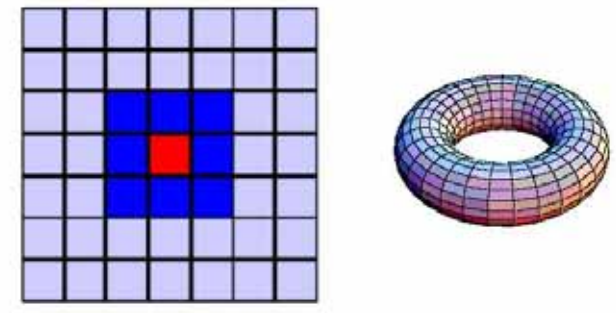
The cellular genetic algorithm has the problem of premature convergence, which is the same as the canonical genetic algorithm. To avoid trapped into local minima, we can use random walk strategy to jump out of the local minima. The mutation operator in GA is some kind similar to random walk, which guarantees the exploration of the search towards different location and might be more promising regions. Gianluigi Folino et al. first combined the cellular genetic algorithm and greedy local search for solving the SAT problem in [9] [10], they call their algorithm as CGWSAT. And they implement CGWSAT on a parallel machine (Meiko CS-2). We want to implement the algorithm on GPU, but we find the original algorithm is not suitable for running on GPU, so we make some modification on it.

The pseudo-code of our modified algorithm performed by each cell is shown below.

```
Algorithm: The cellular genetic algorithm
Let pcross be the crossover probability
for (all i in CGA) {
    //generate a random individual t_i;
    t =generatet();
    //evaluate the fitness of t_i;
    f=f(t_i );
```

```
}
while (not satisfy the stop criterion ){
   for (each cell i in CGA) {
     //generate a random probability;
     cross=cgrand();
     if(cross>pcross) {
        //get best fitness neighbor
        j=select();
        u = crossover (t,t);
        evaluate_fitness(u);
        t = best_fitness(u,t);
     }
     t = WSAT_Strategy(t);
     evaluate_fitness(t);
   }
}
```

The cellular genetic algorithm adopts a 2-dimensional toroidal grid, using the Moore neighborhood (show in figure 1). At beginning, a random population of chromosomes is generated and their fitness is evaluated. At each generation, the partner of each cell, which is among the *k* Moore neighbors of the cell and has the best fitness, is selected. The genetic operator of crossover is then applied with *pcross* probability.



(a) (b)

Fig. 1. （a）Moore neighborhood (b) 2-dimensional toroidal grid

We choose the uniform crossover as the crossover operator. The principle of uniform crossover is show in figure 2. The uniform crossover is also called template crossover. The template is randomly generated, and we get the offspring by copy the corresponding bit of two parents according the template. Uniform crossover is only one kind of crossover operators, and there are many other crossover operators, such as single-point crossover, multi-point crossover, multi-parent crossover etc, which can also be used in our algorithm.

```
Parent.A = 00000000000
Parent.B = 11111111111
Template = 10101011000 /* 1: for A and 0 for B */
Offspring= 01010100111
```

Fig. 2. uniform crossover

For the select operation, we adopt the elitist strategy, letting the offspring compete with the parent and keep the one which has a better fitness value. Bit mutation is performed according to the mixed random walk strategy, which is same as introduced in section II.B. We used the strategy II and III for GPU.

We can see from the pseudo-code of the algorithm, that when the probability of crossover in the algorithm is set to zero, only the mutation operator works. This is a parallel implementation of WSAT, named as PWSAT. It can also improve the efficiency of WSAT algorithm in some degree.

IV. Implementation Details on GPU

The architecture of GPU is SIMD (Single Instruction, Multiple Data). Its input data is usually stored in a 2-D texture, which is same as the population distribution of CGA's grid lattice. So we can naturally implement the CGA's algorithm on GPU, and performs the calculation on each cell in parallel.

But there are a few difficulties to map the CGA algorithm to GPU. The first one is that each pixel of texture can store only 4 components, but the length of individual is usually longer than 4. The second one is that a list is needed to denote the various location of variable in a clause. But we don't have a in GPU, and it is hard to store the list in a texture without knowing the length of a list. The third one is that there is no function to create random number in Cg language, which is very important for genetic algorithms. In this section, we will give our solution to these problems.

A. Data Organization

The initialization of population is performed on CPU by randomly generate N (the size of population) chromosomes. The data should be stored in a texture, and there are three possible methods for storing it. The first one is to store every chromosome in a line of pixel, as shown in figure 3a. The second one is to store every chromosome in a square or rectangle, as shown in figure 3b. The third one is to store the population in a 3D texture.



(a) (b)

Fig. 3. Chromosome store scheme (a) store a chromosome in a line. (b) store a chromosome in a square.

The performance of the first scheme is not good, which mainly because of the GPU's specific architecture. It is possible to store population data in a 3D texture. But it is difficult to change the value, for current GPU's output can only be one or a few 2D texture. If we want to change the 3D texture, we need more passes. So the third solution also has a poor performance. The second method is the best solution currently available. It is suitable for the parallel process of GPU and it is the common method to store large data in texture.

B. Random number generator

We need a random number generator to produce the template for uniform crossover operation. And the random number plays a key role in the process of crossover. But there is no built-in random number generator function in Cg. There are two solutions to solve this problem. The first one is generating random number in CPU and storing them in textures, and fetches it from the texture when needed. This solution is not suitable for our case, which need a matrix of different random numbers in every generation. So we implement a function to create random number in Cg. The algorithm we choose is same as the random function in the standard C library.

The seed is generated on CPU and store in a texture. In each generation, we create new random numbers according the previously generated random numbers, and store them in another texture.

$$R_{t+1}=((R_t * f + c)/d) \bmod m$$

Here the parameters are:

f=1103515245
c=12345
d=65536
m=32768.

The random function implementation in Cg is shown in the following code.

```
void rand(out half4 c :COLOR0,
              half2 coords : TEX0,
          uniform sampler2D randTex)
{
   c = tex2D(randTex, coords);
   c.x = fmod((tc.x*1103515245.0f +12345)
              /65536, 32768);
}
```

C. Store list in texture

The main difficulty to realize the GSAT algorithm on GPU is to find the variable with the MaxDiff. To find the variable, the first thing we need to do is to calculate the Diff value of every variable in a chromosome. We can evaluate every clause to calculate the Diff of a variable, but this is time consuming, for there are only a small part of clauses that contain the variables. To speed up the algorithm, we need to use a list to store the index of clause that contains each variable. So how to store the lists in 2D texture is crucial, especially when the size of list is not fixed.

We use a texture which is big enough to store the lists. The width of the texture is larger than the largest length of the lists. Every row represents a list and the first pixel store the list length.

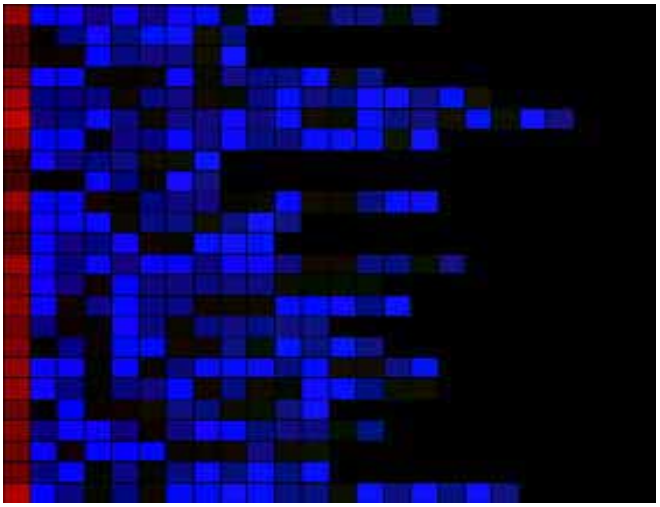


Fig. 4. Store lists in a texture

D. Limitation of GPU architecture

In the process of our experimental, we encounter a few limitation of GPU architecture, the limitation of the width and height of texture. For the old GPU, GeForce 4 and 5, the width and height of a texture must be power of 2. Although for GeForce 6, it supports the texture with width and height are not power of 2, but we find if the width or height of texture is odd, the result is not correct.

Due to this limitation, sometime we may need to use a little bit bigger texture to store data than it is really need. And this will influence the performance of the algorithm.

V. Experimental Result

Our computational environment are P4 2.4G for CPU computing and GeForce 6200 for GPU computing. We use the cellular genetic algorithm discussed above to solve some uniform random-3-SAT problems [13] on both GPU and CPU. The executing results of GPU are similar to that of CPU, but the executing time of GPU is much less than that of CPU.

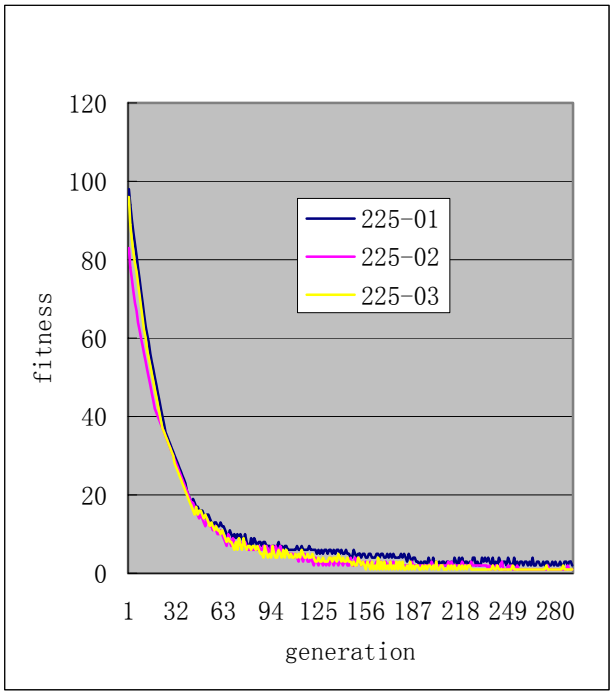
We choose the uf225-960, a set of uniform random-3-SAT problem [13], as examples. There are 225 variables and 960 clauses in the problems. The parameters of the algorithm are chosen as follow, the length of the chromosome is 225, and the size of population is 400.

For comparing the performance of GPU and CPU, we make two specific computation of CGA. One is CGSAT, which takes only crossover but no mutation operator in general CGA. Another is PGSAT (Parallel GSAT), which takes only mutation and no crossover operator in general CGA. The result shows in table 1. From the table we can see a 5-fold performance increase for GPU in both cases.

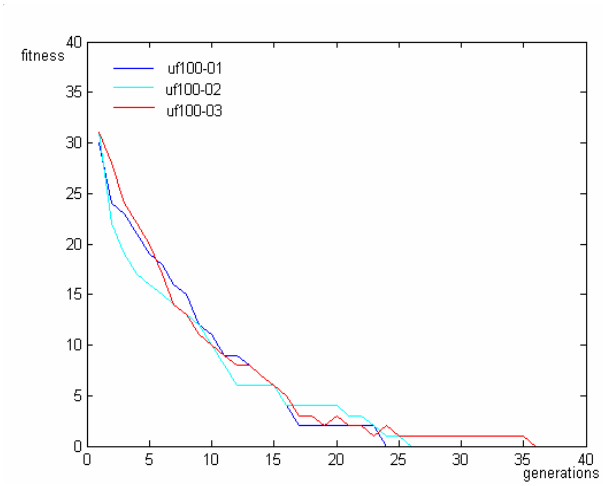
TABLE I
COMPUTING TIME OF CGSAT AND PGSAT

	One Iteration of CGSAT	One Iteration of PGSAT
CPU	95ms	464ms
GPU	18ms	77ms

Figure 5 shows the convergence of our algorithm for two 3-SAT problems on GPU. Figure 5b shows three cases of uf100-430 problem. From the figure, we can see that this algorithm converges fast and can find the solution. Figure 5a shows two cases of uf225-960, the algorithm can't convergence in a short time.



(a)



(b)

Fig. 5. The convergence of uniform random 3 sat problems, (a) uf225-960 (b)uf100-400

VI. Conclusion and Future Work

In this paper, we implement two main algorithms for SAT problems on GPU, and describe the implementation detail using Cg. The inertial parallelism of graphic hardware architecture has made a great acceleration of the greedy local search and cellular genetic algorithm. The experimental result shows that the computation on GPU is much faster than on CPU. The CGA algorithm on GPU can find the solution for the small size (variable number less than 400) problem quickly, and get an approximation solution for large size problem in a short time.

For future works, we hope to explore a better use of our algorithm on more large and hard SAT problems. And we can also use the graphic hardware to other parallel algorithms, and some other soft computing such as swarm intelligence.

It needs some trick to implement the CGA algorithm on a GPU, so we hope to design a general architecture for CGA algorithm on GPU, which will easy the CGA program design on GPU in future computation.

With the fast development of GPU, the programmable GPU will be widely used in more application fields and also become more powerful, which opens a great future for the application of GPU computation.

References

[1] Cook, S. A. The complexity of theorom-proving procedures. Proceedings 3rd Annual ACM Symposium on the Theory of Computing. New York. 1971, 151-158

[2] Gianluigi Folino, Clara Pizzuti , Giandomenico Spezzano, Combining Cellular Genetic Algorithms and Local Search for Solving Satisfiability Problems, 1998

[3] De Jong, K.A. & W. Spears (1989) Using Genetic Algorithms to Solve NP-Complete Problems, International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, June 1989, pgs. 124 - 132.

[4] Stefan Harmeling, Solving Satisfiability Problems with Genetic Algorithms, Genetic Algorithms and Genetic Programming at Stanford 2000, 206-213

[5] Bart Selman ,Henry Kautz, Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiebility Problems, Proceeding of IJCAI-93, 1993

[6] Femando R. and Killgard, M.J. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics Addison-Wisley, 2003

[7] Gianluigi Folino, Clara Pizzuti , Giandomenico Spezzano, Solving the Satisfiability Problem by a Parallel Cellular Genetic Algorithm,1998

[8] LUO Zhongwen, YANG Zhengping, LIU Hongzhi ,LV Weixian, GA Computation of 3-SAT problem on Graphic Process Unit, ISICA'2005

[9] Thompson, C. J., Hahn, S., and Oskin, M. Using modern graphics architectures for general-purpose computing: A framework and analysis. International Symposium on Microarchitecture. 2002.

[10] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," in Proc. 10th Nation Conf. Artificial Intelligence, July 1992, pp. 459–465.

[11] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in Proc. 12th Nation Conf. Artificial Intelligence, 1994, pp. 337–343.

[12] Darrell Whitley, A Genetic Algorithm Tutorial, Statistics and Computing(4):65-85, 1994

[13] SATLIB: The satisfiability library [Online]. Available: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/