Genet Program Evolvable Mach (2009) 10:391–415
DOI 10.1007/s10710-009-9091-4

ORIGINAL PAPER

Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework

Asim Munawar · Mohamed Wahib · Masaharu Munetomo · Kiyoshi Akama

Received: 13 May 2009 / Revised: 25 August 2009 / Published online: 20 October 2009

Abstract General Purpose computing over Graphical Processing Units (GPGPUs) is a huge shift of paradigm in parallel computing that promises a dramatic increase in performance. But GPGPUs also bring an unprecedented level of complexity in algorithmic design and software development. In this paper we describe the challenges and design choices involved in parallelizing a hybrid of Genetic Algorithm (GA) and Local Search (LS) to solve MAXimum SATisfiability (MAX-SAT) problem on a state-of-the-art nVidia Tesla GPU using nVidia Compute Unified Device Architecture (CUDA). MAX-SAT is a problem of practical importance and is often solved by employing metaheuristics based search methods like GAs and hybrid of GA with LS. Almost all the parallel GAs (pGAs) designed in the last two decades were designed for either clusters or MPPs. Unfortunately, very little research is done on the implementation of such algorithms over commodity graphics hardware. GAs in their simple form are not suitable for implementation over the Single Instruction Multiple Thread (SIMT) architecture of a GPU, and the same is the case with conventional LS algorithms. In this paper we explore different genetic operators that can be used for an efficient implementation of GAs over nVidia GPUs. We also design and introduce new techniques/operators for an efficient

A. Munawar (✉) · M. Wahib
Graduate School of Information Science & Technology, Hokkaido University, Sapporo, Japan
e-mail: asim@uva.cims.hokudai.ac.jp

M. Wahib
e-mail: wahibium@uva.cims.hokudai.ac.jp

M. Munetomo · K. Akama
Information Initiative Center, Hokkaido University, Sapporo, Japan

M. Munetomo
e-mail: munetomo@iic.hokudai.ac.jp

K. Akama
e-mail: akama@iic.hokudai.ac.jp

implementation of GAs and LS over such architectures. We use nVidia Tesla C1060 to perform several numerical tests and performance measurements and show that in the best case we obtain a speedup of 25×. We also discuss the effects of different optimization techniques on the overall execution time.

Keywords Compute unified device architecture (CUDA) · General-purpose computing on graphics processing unit (GPGPU) · Genetic algorithm (GA) · MAXimum SATisfiability problem (MAX-SAT) · Single instruction multiple data (SIMD) · Single instruction multiple threads (SIMT)

1 Introduction

The SATisfiability (SAT) problem is a decision problem that asks whether a binary tuple can be found that satisfies all clauses in a Boolean formula. SAT problem is a central problem in theoretical computer science, artificial intelligence, mathematical logic, and many other applications. Its wide range of applications motivates the huge interest shown for this problem. MAXimum SATisfiability problem (MAX-SAT) is a special case of satisfiability problem. It is an NP-hard problem that asks for the maximum number of clauses that can be satisfied by a given assignment.

Given n binary variables $x(j)$, j in N, m clauses C_i, i in M, and weights $w(i)$, where $N = 1, 2, \ldots, n$ and $M = 1, 2, \ldots, m$, the MAX-SAT problem asks to determine a binary tuple (that is, a 0-1 assignment to each of the binary variables) that maximizes the sum of the weights of the satisfied clauses. Each clause is a disjunction of literals $l * (j)$, where a literal is either a variable $x(j)$ (that is, the variable occurs in positive form) or its negation neg $x(j)$ (that is, the variable occurs in negative form). Without loss of generality we assume that at most one of $x(j)$ and neg $x(j)$ is included in each clause. A clause is satisfied if at least one of the positive variables contained in the clause is assigned the value 1 (true) or a negated variable is assigned the value 0 (false).

Algorithms for solving MAX-SAT optimization problem can be divided into two main classes [6]:

1. *Exact or Complete Algorithms*: are dedicated to solve the decision version of SAT problem. The well-known algorithms are based on the Davis-Putnam-Loveland procedure DPL [16]. Satz [12, 13] is a famous example of a complete algorithm. Exact algorithms are usually employed for solving small size problems.
2. *Approximate or Incomplete Algorithms*: are mainly based on Local Search (LS) and Evolutionary Algorithms (EAs). Gsat [21], Tabu search [5, 19], simulated annealing [11], Genetic Algorithms(GAs) [9] and scatter search [7] are all examples of incomplete algorithms for SAT. Hybrids of these algorithms are also in common use. Incomplete algorithms are mostly used to find approximate solution to the problems involving hundreds of variables.

GAs are often used to solve MAX-SAT problems. In their simple form, GAs have a tendency to lose their diversity prematurely, and hence their efficiency for solving MAX-SAT problem. In order to prevent this from happening different techniques are in common use, namely: increasing population size, using sub-populations based GAs [9], employing LS and niching. All these techniques come at an expense of increased computational cost. This is where parallel computing comes for the rescue. For the past two decades clusters/MPPs have been the major parallel architectures. However, in this decade other architectures like multicores and Graphical Processing Units (GPUs) are competing as an affordable and energy efficient alternatives to conventional parallel computing paradigms. Such architectures are becoming more and more common and their importance cannot be ignored anymore.

GPU is a highly parallel multithreaded and manycore processor originally designed for computer graphics. With the addition of programmable stages and higher precision arithmetic, GPUs are now commonly used for applications that were traditionally handled by a CPU. This use of GPU for non-graphics applications is known as General Purpose processing using GPU or GPGPU in short. GPUs are designed such that more transistors are devoted to data processing rather than data caching and flow control. Therefore, GPUs are less general purpose than CPUs. Compute Unified Device Architecture (CUDA) [1] programming model by nVidia is very well suited to expose the parallel capabilities of GPUs through industry standard programming languages. CUDA offers significant innovation and gives users a significant amount of control over a powerful streaming Single Instruction Multiple Threads (SIMT) computer. Since the release of CUDA in 2007, the use of GPGPU has drastically increased in a wide area of applications. Moreover, CUDA is simple to understand as it uses the conventions of C language with some simple extensions.

On the other hand, most of the parallel GAs (pGAs) literature during the last two decades deal with the implementation of pGAs over clusters or MPPs. Unfortunately, very little research is done to explore an efficient implementation of pGAs over GPU like SIMD/SIMT architectures. GA and LS hybrid algorithms are usually suitable for cluster like parallel environments; but, conventional genetic and LS algorithms are not suitable for implementation over GPU like SIMD architectures [23]. Memory management and massive data parallelism make GPUs a challenging environment for algorithm implementation. Nevertheless, GPUs can show unprecedented speedups for an appropriately designed algorithm.

The main motivation behind this paper is to harness the computational power of modern day GPUs to reduce the total execution time for solving MAX-SAT problem using a hybrid of GA + LS without compromising the output fidelity. Utilizing memory bandwidth and massive data parallelism in an efficient way are the major design constraints. Although nVidia's Tesla C1060 is capable of 933 GFLOPs/s of processing performance and has a memory bandwidth of 102 GB/s, reaching peak performance or near peak performance is not an easy task. The objective of this research is to have a GPU compatible GA + LS algorithm, where the LS algorithm and genetic operators are designed to take maximum advantage of the GPU processing power and memory bandwidth. The primary contribution of this research is the modification of conventional LS algorithm and genetic operators to make them suitable for SIMT architecture. We also suggest different kinds of optimizations with empirical results to show their effect on the total execution time. By writing this paper, we also want to encourage more research in the area of metaheuristics based algorithms over modern parallel architectures. We believe that such architectures will become more and more common in the coming years, making similar implementations inevitable in the very near future. We have not only designed and implemented the proposed algorithm, but in the results section (Sect. 6) we empirically demonstrate the speedups achieved by solving well known benchmark instances of MAX-SAT. We also discuss the effect of different optimizations on the overall performance. It involves the optimizations within the algorithm and optimization of the code itself. We believe that our work provides a valuable contribution for application developers, by identifying a path that can be followed by other similar applications.

In order to make this paper self-contained, Sect. 2 describes the relevant architectural features of GPU and CUDA framework. Section 3 describes the past work done to solve MAX-SAT using GAs. We also discuss the past work done in the area of GAs over GPUs. In Sect. 4 we discuss the proposed algorithm including genetic operators and LS algorithms. Section 5 describes the proposed implementation to solve MAX-SAT problem. In this section, we discuss the design of different operators in detail. Section 6 shows the empirical results obtained and compares them with the results obtained by other state-of-the-art CPU architectures. Section 7 concludes this paper and gives some guidelines for future work.

2 General-purpose computing on GPU (GPGPU)

GPU is emerging as one of the most powerful parallel processing devices. GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (i.e. ratio of arithmetic operations to memory operations). As the same instructions are executed for each data element, there is a lower requirement for sophisticated flow control. The memory access latency can be hidden with calculations instead of big data caches. Applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. In the recent years, many algorithms outside the field of image rendering and processing are also accelerated by data-parallel processing. GPUs are increasingly being used for general purpose processing; as a consequence the term GPGPU has evolved.

Although GPUs can offer unprecedented performance gain, implementation of an algorithm over a GPU to take full advantage of this new technology involves a significant complexity of parallelizing across the multiple cores. Memory management over a GPU makes things even more challenging. CUDA is a parallel computing architecture developed by nVidia. CUDA is the compute engine in nVidia's CUDA compatible GPUs, and is accessible to software developers through industry standard programming languages like C. CUDA is widely used for programming nVidia's GPUs for general purpose processing. Research is also done to use CUDA for programming multicore processors other than GPUs [22]. To the

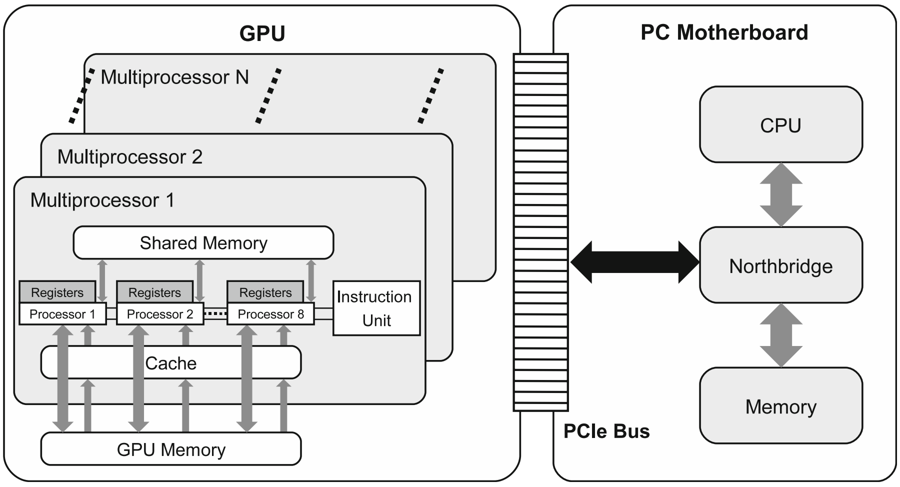

Fig. 1 Hardware architecture of GPU mounted on the PC motherboard [3]

best of author's knowledge, there is no significant research done for implementation of GAs or GA + LS hybrid algorithms over GPU using CUDA.

Figure 1 illustrates the architecture of nVidia Tesla GPU (used for this research) [15]. The GPU runs its own specified instructions independently from the CPU but it is controlled by the CPU. A thread is the computing element in the GPU. When a GPU instruction is invoked, blocks of threads are defined to assign one thread to each data element. Arrangement of blocks and threads in a GPU is shown in Fig. 2. All threads in one block run the same instruction on one streaming MultiProcessor (MP), which gives the GPU an SIMT architecture. Each MP includes 8 stream processor (SP) cores, an instruction unit, and on-chip memory that come in three types: registers, shared memory, and cache (constant and texture). As shown in Fig. 1, threads in each block have access to the shared memory in the MP, as well as to a global memory in the GPU. Unlike a CPU, the internal structure of a GPU is designed in such a way that more transistors are devoted to data processing rather than data caching and flow control. This difference is clearly illustrated in Fig. 3. When an MP is assigned to execute one or more thread blocks, the instruction unit splits and creates groups of parallel threads called warps. The threads in one warp are managed and processed concurrently. Unlike SIMD architecture, threads in the same warp can follow different instruction paths. However, this phenomenon drastically reduces the performance and should be minimized for an efficient implementation.

3 Related work: GA + LS to solve MAX-SAT

A simple GA (sGA) for solving MAX-SAT is shown in Fig. 4. However, sGA suffers from problems like premature convergence and is seldom used to solve complex problems like MAX-SAT. Previous work in the area suggests the use of

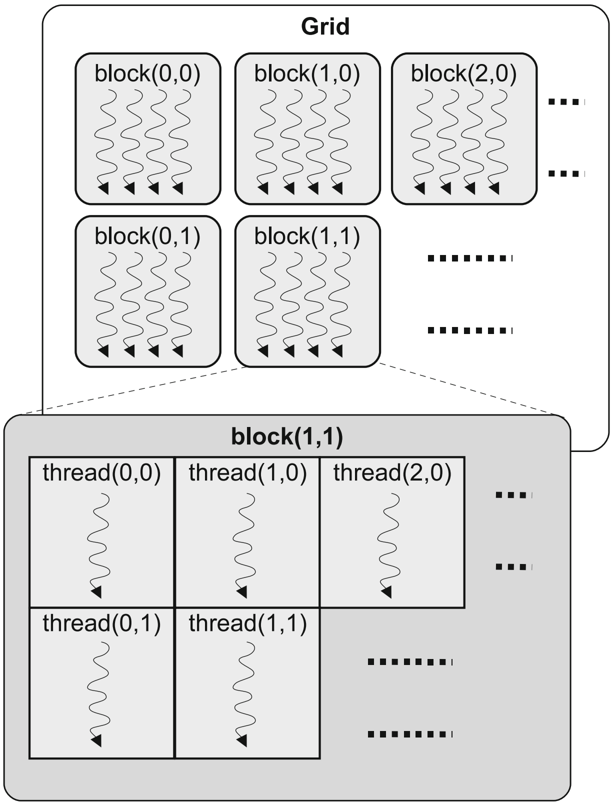

Fig. 2 Hierarchy of computing structure in a GPU [3]

techniques like: larger population sizes, sub populations based pGAs [9] and niching. Moreover, a hybrid of GA and LS has also proved to be an order of magnitude faster than sGA [9, 6].

Frank [9] suggests the use of multiple sub-populations for a GA based MAX-SAT solver. This kind of island based pGAs can easily be implemented over conventional parallel architectures (like clusters). However, implementation over modern parallel architectures like GPU is an entirely different issue. Traditional pGA + LS algorithms and operators (as shown in Fig. 4) are not suitable for implementation over SIMT architectures [23]. Some research have been done in the area of GAs and other EAs over consumer-level graphics hardware or GPUs [8, 24, 14]. Tomassini [23] and Qizhi et al. [24] recommend the use of GA with 2D structured population also called cellular Genetic Algorithm (cGA) [4] for implementation of pGAs over a GPU (SIMT architecture). 2D structure of cGA maps well to the GPU architecture, which is primarily designed for processing of multidimensional graphics.

As far as performance gain is concerned, Fok et al. [8] shows a 1.25 to 5 times speedup over a consumer level GPU card, while Qizhi et al. [24] shows speedups of up to 17.1 times for population sizes of 512^2.

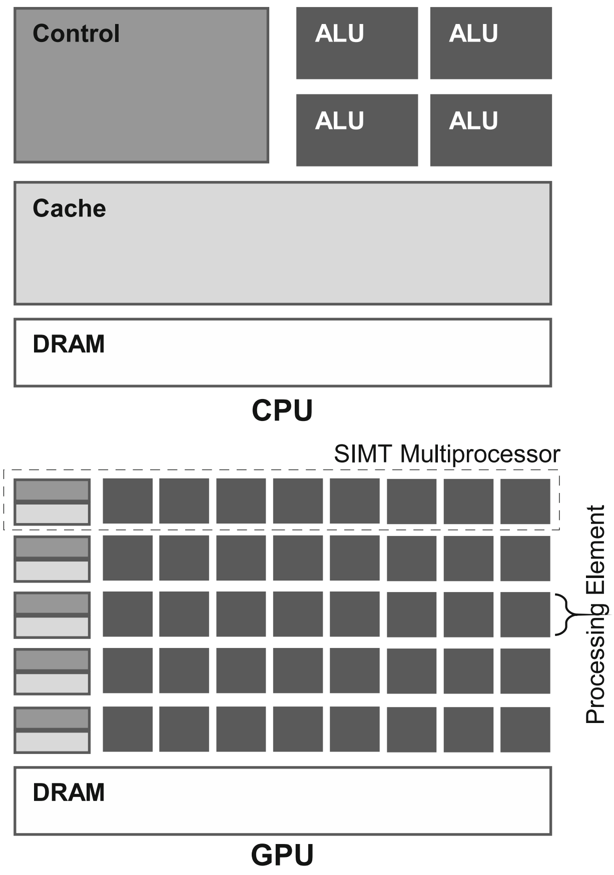

Fig. 3 nVidia GPU Architecture (Note that the GPU devotes more transistors to data processing) [3]

4 Proposed algorithm

Keeping in mind the recommendations made by Frank [9], Tomassini [23] and Qizhi et al. [24], we are using a hierarchical algorithm of 2D structured sub-populations arranged as islands in a 2D grid. Therefore, each individual has 4 neighboring individuals (north, south, east and west) and each sub-population has 4 neighboring sub-populations (north, south, east and west). In CUDA, this kind of hierarchical population arrangement can easily be organized by using 2D blocks and 2D grid as shown in Fig. 2.

In the case of cGA, an individual can only mate with its neighboring (north, south, east and west) individuals [4] as shown in Fig. 5. The individual becomes the first parent while the second parent is selected by applying a binary tournament among the four neighbors. Small overlapped neighborhoods help in exploring the search space because the induced slow diffusion of solutions through the population provides a kind of exploration, while exploitation takes place inside each

```
Input: an instance of satisfiability;
Output: an assignment of variables that maximizes the number of satisfied clauses;

1 : Initialize();
2 : While (max generations or optimal solution is found) do
3 :     For i ← 1 to POP_SIZE; i=i+2 do
4 :         Select two individuals;
5 :         Generate at random a number Rc from [0,100];
6 :         If (Rc <  crossover rate) then apply the crossover;
7 :         Generate at random Rm from [0, 100];
8 :         If (Rm <  mutation rate) then apply mutation
9 :         Evaluate the new individuals;
10:         Apply local search;
11:     End for
12:     Replace the bad individuals of the population by the fittest new ones.
13: End while
14: Return the best chromosome
15: Destroy();
```

Fig. 4 A simple Genetic Algorithm (sGA) to solve MAX-SAT problem (row 10 can be added to an sGA to form a GA + LS hybrid algorithm)

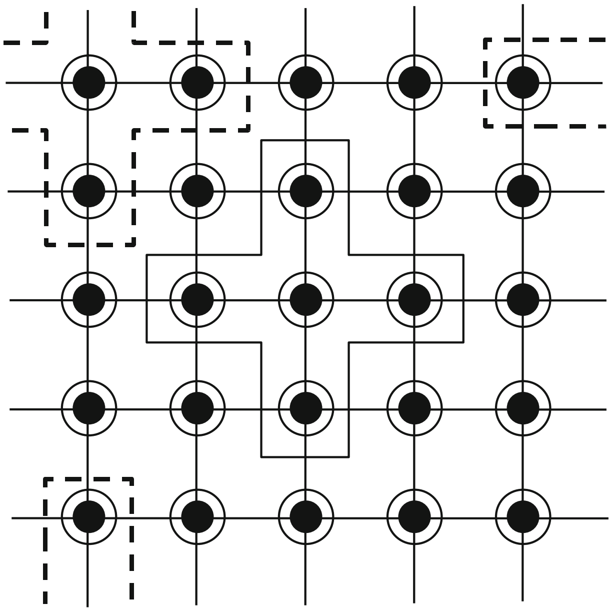

Fig. 5 Population arrangement for cellular Genetic Algorithm (cGA)

neighborhood by genetic operators. As a result, cGA provides automatic niching effect, avoiding an early convergence.

We have used hill-climbing as the LS algorithm. For genetic operators we have used mutation, recombination, and selection. Mutation is simple inversion of a random gene (i.e. 0 → 1, 1 → 0). Recombination is done by using a mask of 0's and 1's as explained in Sect. 5.2.2. For selection, we use binary tournament among the neighbors to select the second parent. Elitist replacement is also performed to insert an offspring into the next population. The fitness function used is the total number of true clauses achieved by a given assignment of the variables. A serial implementation of cGA + LS to solve MAX-SAT is given in Fig. 6.

Instead of using a conventional algorithm for migration between the sub-populations, we have introduced a new technique that we call diffusion. Diffusion is more suitable for implementation of cGAs based pGA over a GPU. Section 5.2.1 explains the proposed diffusion operator in detail.

5 Proposed implementation

Efficient implementation of the proposed algorithm over CUDA compatible GPUs is the main contribution of this paper. In this section we give an efficient implementation of the proposed algorithm (see Sect. 4). The section expounds all the challenges and their solutions. We have modified the existing operators and we also introduce some new operators that can help in increasing the efficiency of the algorithm over a GPU. The final goal is to get the maximum possible speedup without compromising on the solution's quality. The section starts with the overall flow of the implementation and then elaborates each part separately in detail.

In the proposed implementation, the host processor (CPU) acts as a controller while an nVidia Tesla C1060 GPU provides the required computational resources. Figure 7 shows the CPU and the GPU side algorithm. All the configurations, memory allocations, initializations are performed over the host processor. After the initialization stage, data is transferred to the device and the code enters a loop. Inside the loop the first task is to call *kernel 1: random number generator*. In the next step *kernel 2: GA kernel* is deployed over the device. The loop keeps on

```
1 : Initialize_cGA(InputParams);
2 : for s ← 1 to MAX STEPS do
3 :   for x ← 1 to WIDTH do
4 :     for y ← 1 to HEIGHT do
5 :       NList ← Get Neighborhood(Pop(x,y));
6 :       Parents ← LocalSelect(NList);
7 :       AuxIndiv ← Recombination($P_c$, Parents);
8 :       AuxIndiv ← Mutation($P_m$ , AuxIndiv);
9 :       AuxIndiv ← Local Search(AuxIndiv);
10:       EvaluateFitness(aux indiv);
11:       InsertIfBetter(Pop(x,y), AuxIndiv, AuxPop);
12:     end for
13:   end for
14:   Pop ← AuxPop;
15:   UpdateStatistics(Pop);
16: end for
17: Destroy();
```

Fig. 6 Pseudocode for the cGA + LS [4]

repeating until the end criteria is satisfied. In the proposed implementation the loop can only exit when the maximum number of generation criteria is satisfied. After exiting the loop, the final population is copied back to the host memory and the individual with the maximum fitness is selected as the best solution.

5.1 Kernel 1: Random number generator

Current graphics hardware does not provide the function for generating random numbers. Fortunately, CUDA SDK comes with a sample random number generator [20] based on Mersenne twister proposed by Matsumoto et al. [17]. Mersenne twister has properties like long period, efficient use of memory, good distribution properties, and high performance. With some modifications proposed by Matsumoto [18], the algorithm maps well onto the CUDA programming model to produce uniformly distributed pseudo-random numbers. As shown in Fig. 7, the output of the random number generation kernel is Nr random numbers. Nr is equal to the total number of random numbers required to evaluate one generation. This includes the random numbers required by mutation, crossover, and selection stages. Nr can be given as follows:

$$Nr = Nr_s + Nr_c + Nr_m$$

where, Nr_s, Nr_c, Nr_m are the total number of random numbers required by selection, recombination, and mutation, to process a single generation (for details

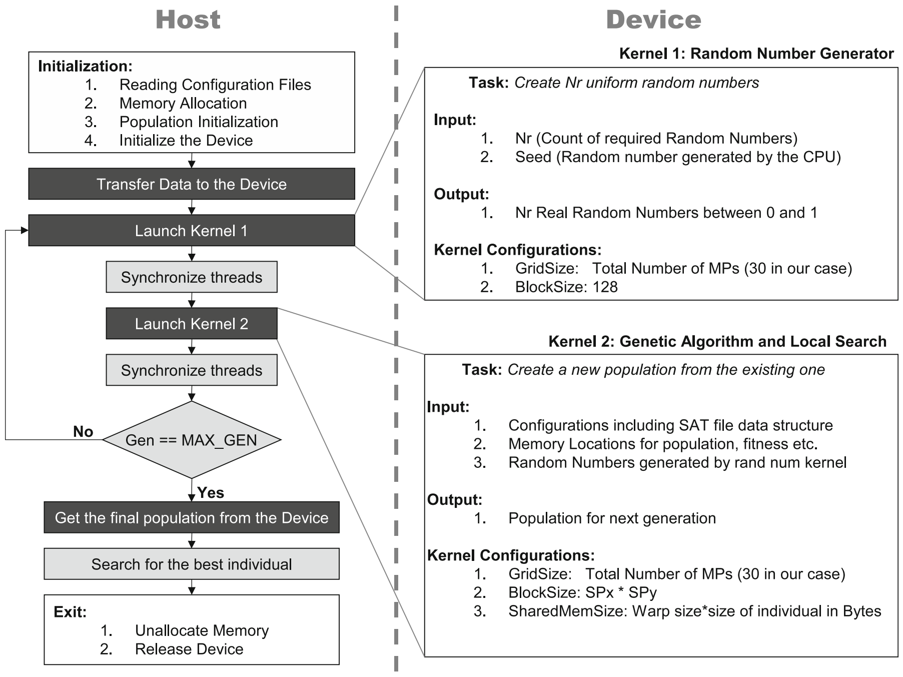
Host
Initialization:
1. Reading Configuration Files
2. Memory Allocation
3. Population Initialization
4. Initialize the Device
Transfer Data to the Device
Launch Kernel 1
Synchronize threads
Launch Kernel 2
Synchronize threads
Gen == MAX_GEN
No
Yes
Get the final population from the Device
Search for the best individual
Exit:
1. Unallocate Memory
2. Release Device
Device
Kernel 1: Random Number Generator
Task: Create Nr uniform random numbers
Input:
1. Nr (Count of required Random Numbers)
2. Seed (Random number generated by the CPU)
Output:
1. Nr Real Random Numbers between 0 and 1
Kernel Configurations:
1. GridSize: Total Number of MPs (30 in our case)
2. BlockSize: 128
Kernel 2: Genetic Algorithm and Local Search
Task: Create a new population from the existing one
Input:
1. Configurations including SAT file data structure
2. Memory Locations for population, fitness etc.
3. Random Numbers generated by rand num kernel
Output:
1. Population for next generation
Kernel Configurations:
1. GridSize: Total Number of MPs (30 in our case)
2. BlockSize: SPx * SPy
3. SharedMemSize: Warp size*size of individual in Bytes

Fig. 7 Flow chart of CPU (*host*) side and GPU (*device*) side logic

see Sects. 5.2.1, 5.2.2 and 5.2.3). Value of *Nr* is computed on the host and is provided as an input to the kernel along with the seed *S*. In order to avoid correlation among random numbers generated in each iteration *S* should be random itself. For each generation *S* is computed using standard C language routines for random number generation. This is ultimately seeded with the system time. As both random number generators use different algorithms therefore the risk of correlation is minimized.

For random number generator kernel gridSize is equal to the total number of MPs (30 in our case). Number of threads is kept at 128 threads/block to achieve a device occupancy[1] of "1". It is important to note that the random numbers generated by this kernel are stored in the global memory of the device and are never transferred to the host memory.

5.2 Kernel 2: Genetic algorithm kernel

Genetic algorithm kernel is the heart of the proposed implementation. For an efficient implementation of this kernel, we must keep the following facts about the CUDA and GPU architecture into consideration:

1. Each thread should solve a very small part of the problem. This helps in increasing the total number of threads and hence utilizing the resources in a better way.
2. Minimize the use of global device memory. Shared memory and registers should be used wherever possible.
3. Global memory access should be coalesced wherever possible.
4. Threads in the same warp should avoid branching. Note that all the threads in a warp are controlled by a single instruction unit.
5. Try to maximize occupancy. Occupancy can be maximized by choosing an appropriate value thread block size, shared memory size and number of registers used.

The proposed implementation is similar to a fine-grain genetic algorithm, as each thread deals with only one individual. Therefore, the total number of threads is equal to the population size. The population is arranged into a number of sub-populations. Each sub-population ultimately runs on one MP. Hence, total number of sub-populations is equal to the number of MPs available in the GPU. Each population is arranged in a 2D toroidal shape. Each block is of size $SP_x * SP_y$ where, SP_x and SP_y are the x-dimension and y-dimension of the block, respectively.

Flowchart diagram of *kernel 2* is shown in Fig. 8. Each part is designed very carefully to get an overall efficient implementation. As shown in the figure, the kernel is divided into six distinct parts (shown by white boxes in Fig. 8). It will only be fair to discuss each of these parts separately with implementation/optimization details.

[1] Occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU, and is helpful in determining how efficient the kernel will be on the GPU.

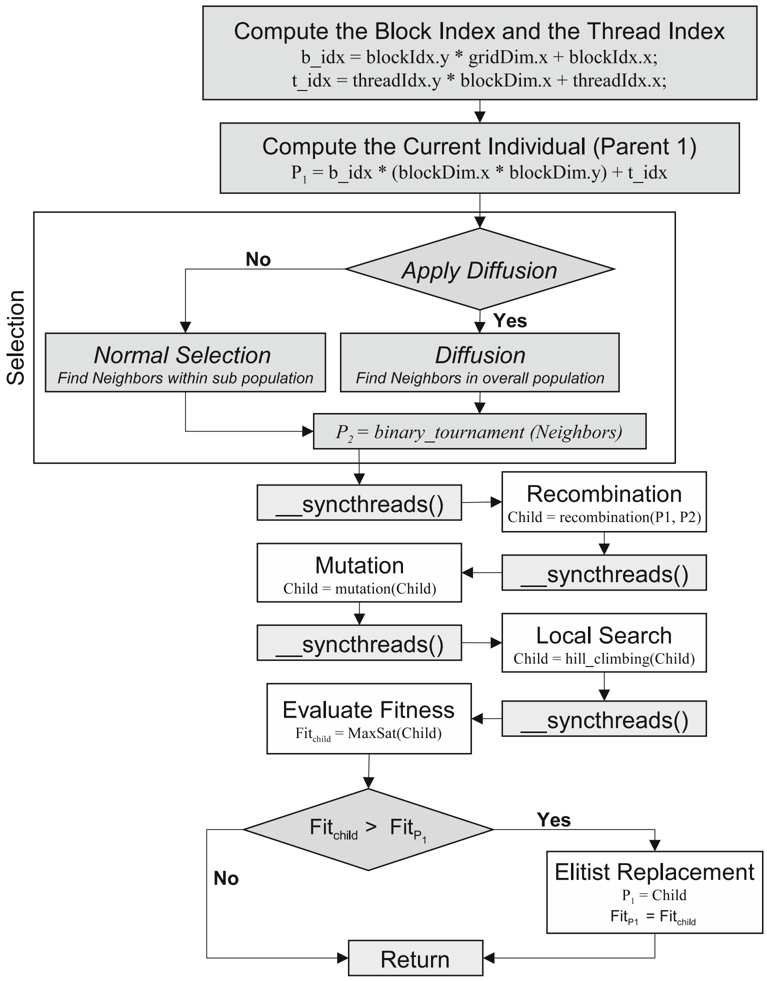
Compute the Block Index and the Thread Index
b_idx = blockIdx.y * gridDim.x + blockIdx.x;
t_idx = threadIdx.y * blockDim.x + threadIdx.x;
Compute the Current Individual (Parent 1)
P1 = b_idx * (blockDim.x * blockDim.y) + t_idx
Selection
Apply Diffusion
No
Yes
Normal Selection
Find Neighbors within sub population
Diffusion
Find Neighbors in overall population
P2 = binary_tournament (Neighbors)
__syncthreads()
Recombination
Child = recombination(P1, P2)
Mutation
Child = mutation(Child)
Local Search
Child = hill_climbing(Child)
Evaluate Fitness
Fitchild = MaxSat(Child)
Fitchild > FitP1
Elitist Replacement
P1 = Child
FitP1 = Fitchild
Return

Fig. 8 Kernel 2: is the critical kernel that performs Genetic Algorithm and Local Search operations. This kernel is launched once in each generation and overall it takes more than 95% of the total execution time

5.2.1 Selection and diffusion

In a typical island model (sub-population based) pGA, selection and migration takes place as two independent steps. Selection is applied within a sub-population while migration is responsible to exchange individuals having good fitness with other sub-populations. Tournament selection is a very common type of selection operator and migration often involves operations like sorting or any other similar algorithm. Such operations involve branching, making them unsuitable for implementation over a GPU. In this paper we have merged the selection and migration steps to create a new kind of operator that we call “Selection and Diffusion”. In diffusion the selection operator is used to share the individuals at the borders of a sub-population with their neighboring sub-population. This new operator is suitable for efficient implementation over a GPU with solution quality comparable to its conventional counterparts.

As discussed in Sect. 4, we are using cGA as the base of the proposed algorithm. Each sub-population runs independently over a separate MP as shown in Fig. 9. The neighborhood always contains 4 individuals: North, East, West, and South individuals. The individual of the thread becomes the first parent while the second parent is selected by using binary tournament among the neighboring individuals. Under normal circumstances selection is applied and the neighborhood is defined within a sup-population as shown in *“sub pop 1”* of Fig. 9. However, with a probability P_d, diffusion is applied to the whole population in that generation. When, diffusion is applied the neighborhood is defined as shown in *“sub pop 5”* of Fig. 9. As the whole population resides in the global memory of the device therefore, all the data is accessible by any thread in any block.

We can control the migration rate of individuals by controlling the diffusion probability P_d. If P_d is 0, all the sub-populations are completely independent and they do not exchange any individual with their neighboring sub-populations. On the other hand, if P_d is 1, whole population acts as a single entity. Under normal circumstances, P_d should be kept closer to 0 but not 0. This will limit the number of individuals diffusing into the neighboring sub-populations. The reason why diffusion works is that the relatively small 2D structured sub-populations reduce the distance of an individual from their neighboring sub-populations, hence making it easier for an individual with higher fitness to diffuse into the neighboring sub-populations. Therefore, diffusion cannot work with non-structured populations.

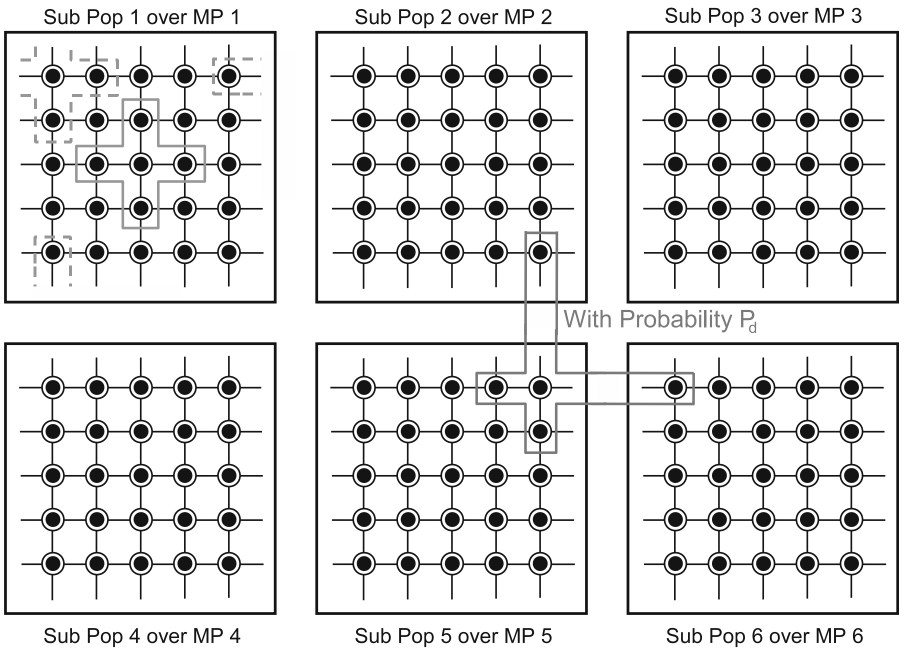

Fig. 9 Selection and Diffusion operator

As each individual requires 2 random numbers for a binary tournament among its neighbors, Nr_s is equal to twice the size of the population (as mentioned earlier, these random numbers must be generated by kernel 1). Selection and diffusion operator does not involve any looping and has only one *conditional statement* without an *else* statement.

5.2.2 Recombination

After the selection stage, we have two parent individuals $P1$ and $P2$, where, $P1$ is the current individual and $P2$ is the neighboring individual selected by the selection/diffusion operator (Sect. 5.2.1). Recombination combines these two parents to produce a single child individual C. We are using the shared memory to store the child's data, because the remaining operators namely: mutation, fitness evaluation, LS and replacement are applied to the child individual (as shown in Fig. 8). Therefore, keeping child's data in the shared memory can have a significant impact on the overall execution time. But increasing the size of shared memory can result in lowering the GPU occupancy. Consequently, it is recommended that the child's data must be kept outside the shared memory for the problems involving larger number of variables.

The recombination operator used in this research is slightly different from a traditional crossover operator. Traditionally, crossover is applied to the individuals with a certain probability P_c. In our case the recombination operator is applied to all the individuals. In order to avoid *conditional statements* involved in one-point or two-point crossover, we perform the crossover using a mask (M) of 1's and 0's. Probability of 1's in the mask is given by P_c. The child C can be computed as follows:

$$C_i = (P1_i \wedge \neg M_i) \vee (P2_i \wedge M_i)$$

where, $i = 1, ..., N$ and N is the total number of variables. This implies, when $M = 0$ then $C_i = P1_i$ and when $M = 1$ then $C_i = P2_i$. Therefore, in absence of any 1 in the mask, $C = P1$, therefore, it will have an effect of no crossover. Conventional crossover operators involve branching. As discussed earlier divergent threads in a warp underutilizes the GPU and must be avoided. In a mask based recombination operator all the threads in a warp follows the same instruction path, hence making it more suitable for implementation over GPU like architectures. In terms of quality, empirical results prove that the recombination operator used in this research is as good as its conventional counterparts.

Number of random numbers required by each individual to perform recombination is equal to the total number of variables N. Therefore, total number of random numbers required by the recombination step Nr_c is N times the population size.

5.2.3 Mutation

Mutation is a bit inversion operator applied to the child C. Mutation is applied by using a mask M of 0's and 1's. Probability of 1's in the mask is given by P_m. Bit inversion is performed by XORing the individual C and the mask M.

$$C_i = C_i \oplus M_i$$

where, $i = 1, ..., N$ and N is the total number of variables. This implies that, the bit is inverted if $M_i = 1$. Mutation operator is a simple operator without any branch statements. Total random numbers required by each individual to perform mutation is equal to the total number of variables N. Therefore, the total number of random numbers required by the mutation step for the whole population Nr_m is N times the population size.

5.3 Fitness evaluation

The fitness value is simply the number of true clauses in any given variable assignment. In every generation, fitness value is evaluated several times for the child in each thread. LS is also dependent on this routine, therefore, efficiency of this routine is an important design parameter. Fortunately, MAX-SAT problem is SIMD/SIMT compatible and all the threads follow same instruction sequence. Fitness evaluation use nested loop with conditional statements in every iteration. As all the threads follow the same branching pattern, therefore, in this case branching does not have a negative effect on the overall performance of the implementation.

5.3.1 Local search (LS)

Among all the operators, LS is the most difficult operator to be implemented over SIMT architecture. The target of this operator is to find the local optima. It is mandatory to optimize this operator, as LS is usually the most time consuming part of the GA + LS hybrid algorithm (see Sect. 6.1.1 for details). As discussed earlier, we are using hill-climbing as the LS algorithm. We have modified the algorithm to meet the hardware constraints for an efficient implementation. A comparison between the typical hill climbing and the proposed algorithm is shown in Fig. 10. In a typical implementation, the maximum number of LS iterations M_{iter} (shown as "MAX_ITER" in Fig. 10) is set to a certain value. If there is an iteration without any fitness gain the algorithm exits right away without waiting for the M_{iter} to occur. In the case of the GPU based implementation, we want all the threads in a warp to exit at the same time and follow a similar execution pattern. Therefore, in case of the proposed implementation LS is performed F_{iter} number of times (shown as "FIXED_ITER" in Fig. 10). A thread cannot *Exit* in the middle and has to finish F_{iter} number of iterations. But keeping a constant value of F_{iter} throughout the algorithm degrades the performance of LS. Therefore, we have developed a feedback system to control the value of F_{iter} and to solve the problem of performance degradation.

In the proposed system F_{iter} is initialized with "MAX_ITER" in the start of the algorithm. With each generation we decrease the value of F_{iter} by an amount "DEC_ITER". On the other hand, each call to LS routine returns a feedback value. This value contains the total number of iterations that were required to reach the local optima. If the feedback value of more than $F_p\%$ of the threads is equal to F_{iter}, it implies that we need to increase the number of F_{iter} in the next generation. Therefore, we incease F_{iter} by "DEC_ITER". Note that the maximum value for F_{iter}

```
MaxFit ← Fitness ← ObjFunc(C);
for i ← 1 to MAX_ITER do
    for idx ← 1 to N do
        $C_{idx} \leftarrow \neg C_{idx}$;
        TempFit ← ObjFunc(C);
        if(TempFit > MaxFit) do
            MaxFit ← TempFit;
        else
            $C_{idx} \leftarrow \neg C_{idx}$;
        endif
    endfor
    if (MaxFit > Fitness)
        Fitness ← MaxFit;
    else
        Break;
    endif
endfor
```

```
MaxFit ← ObjFunc(C);
for i ← 1 to FIXED_ITER * N do
    idx ← i% N;
    $C_{idx} \leftarrow \neg C_{idx}$;
    TempFit ← ObjFunc(C);
    if(TempFit > MaxFit) do
        MaxFit ← TempFit;
        FeedBackIteration ← i / N;
    else
        $C_{idx} \leftarrow \neg C_{idx}$;
    endif
endfor
```

Fig. 10 Hill-Climbing local search operator: Traditional hill climbing algorithm (*left*), Proposed implementation (*right*)

is “MAX_ITER”. This feedback mechanism gives us the required performance and SIMTization without any compromise on the quality of the end results. In our experience, nested loops significantly affect the performance of the kernel. Therefore, we have avoided nested loops as shown in Fig. 10.

5.3.2 *Replacement*

We have used the elitist replacement, i.e. the parent is replaced with the child only if the child has a better fitness than the corresponding parent individual. It is in this step that the child’s data is transferred from the shared memory to the global device memory.

6 Results

Results given in this section were collected over a system with nVidia Tesla C1060 GPU mounted on a motherboard with Intel® *Core*^{TM} i7 920@ 2.67 GHz as the host CPU. C1060 have 4 GB of device memory, 30 streaming MultiProcessors (MPs), and the total number of processing cores is 240. The maximum amount of shared memory per block is 16 KB and clock rate is 1.30 GHz. The compute capability of the device is 1.3. We are using Fedora Core 8 as the operating system and CUDA SDK/Toolkit ver. 2.1 with nVidia driver ver. 180.22. Other tools used for optimization and profiling include *CudaVisualProfiler ver. 1.1* and *CudaOccupancyCalculator*. C1060 is dedicated to computations only. System has a separate GeForce 8400 GS GPU acting as a display card.

The values of P_c & P_m are kept constant at 20 and 10%, respectively. Population size is also kept constant at 3000. For GPU implementation the population is divided into 30 sub-populations each consisting of 100 individuals. 30 sub-populations are spatially arranged in a 2D grid of 10 * 3. Individuals in each sub-population are further arranged in a 2D toroidal of size 10 * 10. In order to keep the comparison fair, we have used sub-populations based pGA even for serial and OpenMP based implementations. Number of sub-populations is kept constant at 30 for all the implementations. For LS maximum number of hill-climbing iterations (MAX_ITER) is kept constant at 20, while the value is decremented by 2 (DEC_ITER) in each generation. Diffusion probability P_d is kept constant at 5%. Feedback percentage F_p is kept constant at 20%. In order to test our results we have used the famous benchmarks from SATLIB [10] (SAT online resources).

6.1 Performance optimization

On top of algorithm parallelization over GPU, we have used several optimization techniques to fine tune the performance of the algorithm and to make it more suitable for the low lying hardware. In this section, we briefly explain these techniques, and evaluate them in terms of their impact on the overall performance.

As a starting point for the experiment, we ported the proposed algorithm (Sect. 4) over C1060 GPU. Population was broken into 30 sub-populations each running as a separate block on a different multiprocessor. Each block contains 100 threads (or individuals). Although parallelization of the algorithm gives us a huge performance gain, it can be further optimized to get even better speedups. We break the optimization step into two passes. In the first pass we try to optimize the implementation by making some changes in the algorithm itself. In the second optimization pass we apply some hardware specific optimization techniques without making any changes in the algorithm. The target is to reduce the total execution time without compromising on the solution's quality.

6.1.1 Optimization pass I

In the beginning of *optimization pass I*, we implemented the proposed algorithm in parallel over GPU. The total execution time taken by this implementation to solve "uf250-01" problem is shown by the first row of Table 1. Although in this implementation the algorithm is running on the GPU, it is not at all optimized for the hardware. We attempted to optimize the algorithm by modifying each stage of the algorithm in order to make it more suitable for the SIMT architecture of the GPU. Reduction in execution time for each optimization step is shown in Table 1. To obtain the results shown in Table 1, we have broken our GA kernel into smaller kernels with each kernel performing only one operation on the population. We then use CUDA timer functions to compute the time taken by each operation separately. As there is no data transfer between GPU and CPU after each operation, breaking the GA kernel into many smaller kernels does not have any effect on the overall execution time. Different stages of optimization are given below:

1. *Optimizing local search*: LS was optimized by keeping the number of iterations constant for each thread in a warp. This reduced the branching and waiting in

Table 1 Optimization Pass I. Performance impact of various optimization steps when applied to "uf250-01" MAX-SAT instance

Algorithm stages	Average execution time for different stages of the algorithm (s)						
	Local search	Mutation	Recombination	Selection	Migration	Others	Total
Optimization steps							
Algorithm over GPU	65.910	3.805	1.791	5.312	3.983	0.912	82.713
Optimizing local search	36.174	3.805	1.791	5.312	3.983	0.912	52.977
Optimizing mutation	36.174	0.092	1.791	5.312	3.983	0.912	49.264
Optimizing recombination	36.174	0.092	0.143	5.312	3.983	0.912	47.616
Optimizing selection	36.174	0.092	0.143	0.353	3.983	0.912	42.657
Optimizing migration	36.174	0.092	0.143	0.353	0.087	0.912	37.761

The algorithm only stops when the maximum number of generations (20 in this case) is reached. All the results shown in this table are an average of 50 independent runs

different threads of the same warp. Feedback system was introduced to cater for the quality loss (see Sect. 5.3.1 for details).

2. *Optimizing mutation*: Instead of using conditional statements, mutation now uses binary xor with a mask for its execution. Mutation is applied to all the individuals.
3. *Optimizing recombination*: From a 2-point crossover we changed the algorithm to use a crossover that uses binary mask for its execution. Crossover is also applied to all the individuals without any exception.
4. *Optimizing selection*: In the last step we optimized the selection operator. It is in this step that the population is organized into 2D toroidal shape. Tournament selection is also changed to a binary tournament selection among the four neighboring nodes.
5. *Optimizing migration*: As discussed in Sect. 5.2.1, we have introduced diffusion, a GPU friendly way to perform migration. In diffusion the individuals at the border of a sub-population diffuse into neighboring sub-populations with a certain probability P_d. Due to relatively small 2D sub-populations diffusion doesn't compromise the result quality. Diffusion can save the time for data transfer and sorting.

It is clear from Table 1, that the optimizations applied have a significant effect on the overall execution time.

6.1.2 Optimization pass II

Although after *optimization pass I* the algorithm is optimized and is suitable for GPU, additional optimizations can be done to further reduce the total execution time. We applied a second pass of optimization using different optimization techniques specific to GPU architecture. We applied these optimization techniques to further optimize the problem discussed in Sect. 6.1.1 (i.e. solving "uf250-01" MAX-SAT instance with number of generations constant at 20). Techniques applied in this optimization pass include:

1. *Branch prevention & branch prediction*: Branch prevention has significant effect on the total execution time. Preventing branching decreases the number of divergent threads in a warp. We applied branch prevention wherever possible, especially for the LS hill climbing operator. Branch prediction is another way to optimize branch operations. It is often used in the modern processors to improve performance. However, in case of a GPU, branch prediction can only be useful if all the threads in a warp follow the same branching pattern. We have not used branch prediction as the threads of the same warp do not follow the same branching pattern in the proposed implementation. Using branch prevention we were able to reduce the total execution time for solving "uf250-01" MAX-SAT instance from 37.761 to 35.486 s.
2. *Optimizing memory transfers*: This step involves several kinds of optimizations. In the first step we reduced the transfers between host and device memory. Now the only data transferred from device to host in each generation is the feedback value of LS which is only 1-Byte per individual. No other data is transferred between host and device throughout the execution. As a next step of memory optimization, we are now using shared memory as a kind of cache for the global device memory. Global memory can be 150$\times$ slower than shared memory or registers. We store all the intermediate data, including the data of the temporary child individuals, in the shared memory. Shared memory is almost as fast as the registers but **_syncthreads()** must be called to ensure the write to the shared memory. In order to further reduce memory overheads, the access to global memory should also be coalesced. In current implementation the parent selection is somewhat random (based on binary selection among the neighbors); hence memory accesses cannot be coalesced in all cases. Optimizing memory transfers resulted in a reduction of the total execution time from 35.486 to 34.118 s.
3. *Increasing occupancy*: We have used nVidia's *CudaOccupancyCalculator* tool to explore the trade-offs between number of threads-per-block versus the number of registers and the amount of shared memory used by each thread. Finding just the right combination of the three values to maximize the occupancy had a significant effect on the execution time of the proposed kernel. By increasing the occupancy we were able to reduce the total execution time from 34.118 to 31.926 s.
4. *Avoiding nested loops*: Although not mentioned clearly in any of CUDA's documents, we found that avoiding nested loops can result in a huge performance gain. We tried to avoid the use of nested loops in the LS algorithm and observed a drastic reduction in execution time. Avoiding nested loops helped in reducing the total execution time from 31.926 to 28.913 s.

Therefore, applying the above mentioned steps in second pass of optimization helped in reducing the execution time to solve "uf250-01" MAX-SAT benchmark instance from 37.761 to 28.913 s.

6.2 Empirical results

For comparison purposes, we have used OpenMP [2] based parallel implementation of the same algorithm over different state-of-the-art commodity processors. All the implementations are optimized for the hardware they are running on. Moreover, for all the implementations we are using the compiler directives to maximize the execution speed. Five different implementations used to obtain the results are shown in Table 2.

Tables 3, 4 and 5 show the average value and standard deviation of the total execution time, number of generations and the best solution obtained by each implementation. The results shown in these tables are an average of 50 independent runs. All the benchmark problems used for the experiments are fully satisfiable. The algorithm exits if the best solution (total number of clauses) is found or if the fitness does not change for 5 consecutive generations. Although this paper proposes different changes in the traditional GA + LS algorithm for implementation over a GPU (P_{gpu}), it is clear from Tables 4 and 5 that these modifications does not affect the solution's quality and the convergence rate.

Figure 11 shows the speedups achieved with respect to each implementation. For larger problems, we can see a maximum speedup of up to 25$\times$ if compared with the serial implementation over Intel Core 2 Duo 3.3 GHz (S_{duo}). For smaller problems with 20 and 50 variables we can still see significant speedups of 16$\times$ and 18$\times$, respectively. Comparing our results with an OpenMP based parallel implementation over Intel Core 2 Duo 3.3 GHz (P_{duo}), we got speedups of up to 13$\times$. Even comparing our results with state-of-the-art processors like Intel Core i7 2.67 GHz 4 cores/8 threads (P_{i7}) and Sun Ultra Sparc 1.167 GHz 8 cores/64 threads (P_{u_sparc}) yielded a maximum speedup of up to 6$\times$ and 8$\times$, respectively. These results clearly show the significance of using GPUs in the area of evolutionary computations.

7 Conclusions and future work

MAX-SAT is an NP-hard optimization problem with practical uses in many areas of science and technology. GAs can be used to solve MAX-SAT problems. However, simple GAs suffer from problems like early convergence. Techniques like Local Search (LS), niching etc are often used to increase the efficiency of GAs to solve

Table 2 Different implementations used to obtain the results

S_{duo}	Serial implementation over Intel®*Core*™ 2 Duo E8600@ 3.33 GHz CPU (2 cores/2 threads) with 4 GB of memory
P_{duo}	OpenMP based parallel implementation over Intel®*Core*™ 2 Duo E8600@ 3.33 GHz CPU (2 cores/2 threads) with 4 GB of memory
P_{u_sparc}	OpenMP based parallel implementation over Sun Ultra Sparc T2 1.16 GHz CPU (8 cores/64 threads) with 16 GB memory
P_{i7}	OpenMP based parallel implementation over Intel®*Core*™ i7 920@ 2.67 GHz CPU (4 cores/8 threads) with 4 GB memory
P_{gpu}	Implementation of the proposed algorithm over nVidia C1060 GPU

Table 3 Average execution time required for solving MAX-SAT problem over different architectures

Benchmarks		Total variables	Total clauses	Average execution time (s) ± SD				
Set	Number			S_{due}	P_{due}	P_{u_spare}	P_{i7}	P_{gpu}
uf20-91	uf20-01	20	91	0.227 ± 0.00138	0.1206 ± 0.00073	0.075 ± 0.00048	0.054 ± 0.00031	0.0144 ± 8.66e-05
uf50-218	uf50-01	50	218	1.837 ± 0.622	0.9 ± 0.31	0.602 ± 0.204	0.3467 ± 0.1174	0.098 ± 0.0335
uf75-325	uf75-01	75	325	4.87 ± 1.67	2.24 ± 0.767	1.35 ± 0.463	0.906 ± 0.31	0.221 ± 0.076
uf100-430	uf100-01	100	430	67.977 ± 14	36.23 ± 7.5	20.66 ± 4.21	13.16 ± 2.68	2.995 ± 0.638
uf125-538	ufl25-01	125	538	110.148 ± 31.43	57.16 ± 16.313	31.14 ± 8.889	22.308 ± 6.4	4.67 ± 1.332
uf150-645	uf150-01	150	645	201.625 ± 59.05	105.4523 ± 30.89	60.74 ± 17.8	43.02 ± 12.68	8.43 ± 2.46
uf175-753	uf175-01	175	753	225.08 ± 64.4	114.85 ± 32.845	66.96 ± 19.15	46.96 ± 13.43	13.43 ± 13.43
uf200-860	uf200-01	200	860	379.35 ± 78.66	193.49 ± 39.11	114.81 ± 23.15	85.39 ± 17.2	15.19 ± 3.12
uf225-960	uf225-01	225	960	548.83 ± 149.07	281.76 ± 76.11	168.53 ± 43.55	123.42 ± 33.41	21.607 ± 5.49
uf250-1065	uf250-01	250	1065	637.42 ± 131.38	328.135 ± 66.13	194.2 ± 40.28	142.37 ± 29.211	24.803 ± 5.14

Table 4 Average number of generations required for solving MAX-SAT problem over different architectures

Benchmarks		Total variables	Total clauses	Average number of generations ± SD				
Set	Number			S_{due}	P_{due}	P_{u_spare}	P_{i7}	P_{gpu}
uf20-91	uf20-01	20	91	1 ± 0	1 ± 0	1 ± 0	1 ± 0	1 ± 0
uf50-218	uf50-01	50	218	1.12 ± 0.1385	1.18 ± 0.48	1.08 ± 0.26	1.1 ± 0.3	1.08 ± 0.26
uf75-325	uf75-01	75	325	1.16 ± 0.447	1.14 ± 0.447	1.12 ± 0.384	1.14 ± 0.447	1.18 ± 0.56
ufl100-430	uf100-01	100	430	8.2 ± 1.711	8.36 ± 1.87	8.08 ± 1.83	8.04 ± 1.665	8.04 ± 1.678
uf125-538	uf125-01	125	538	8.675 ± 2.712	8.8 ± 2.69	8.91 ± 2.74	8.8 ± 2.719	8.85 ± 2.67
uf150-645	uf150-01	150	645	10.35 ± 2.95	10.35 ± 2.933	10.375 ± 2.949	10.4 ± 2.942	10.25 ± 2.99
uf175-753	uf175-01	175	753	10.925 ± 4.68	10.925 ± 4.693	10.875 ± 4.654	11 ± 4.624	11.1 ± 4.64
uf200-860	uf200-01	200	860	12.425 ± 2.26	12.4 ± 2.329	12.525 ± 2.276	12.35 ± 2.56	12.525 ± 2.276
uf225-960	uf225-01	225	960	13.567 ± 3.34	13.8 ± 3.33	13.6 ± 3.41	13.7 ± 3.42	13.467 ± 3.32
uf250-1065	uf250-01	250	1065	17.05 ± 2.46	17.1 ± 2.519	16.925 ± 2.33	16.975 ± 2.48	17.05 ± 2.195

Table 5 Average solution quality obtained by solving MAX-SAT problem over different architectures

Benchmarks		Total variables	Total clauses	Average number of satisfied clauses ± SD				
Set	Number			S_{due}	P_{due}	P_{u_spare}	P_{i7}	P_{gpu}
uf20-91	uf20-01	20	91	91 ± 0	91 ± 0	91 ± 0	91 ± 0	91 ± 0
uf50-218	uf50-01	50	218	218 ± 0	218 ± 0	218 ± 0	218 ± 0	218 ± 0
uf75-325	uf75-01	75	325	325 ± 0	325 ± 0	325 ± 0	325 ± 0	325 ± 0
uf100-430	uf100-01	100	430	428.58 ± 0.609	428.62 ± 0.632	428.6 ± 0.605	428.54 ± 0.5422	428.64 ± 0.597
uf125-538	uf125-01	125	538	535.75 ± 0.6698	535.85 ± 0.769	535.7 ± 0.648	535.8 ± 0.88	535.9 ± 1.008
uf150-645	uf150-01	150	645	642.75 ± 0.438	642.825 ± 0.5	642.75 ± 0.493	642.75 ± 0.4385	642.7 ± 0.51
uf175-753	uf175-01	175	753	749.775 ± 0.659	749.875 ± 0.822	749.76 ± 0.843	749.9 ± 0.841	749.875 ± 0.822
uf200-860	uf200-01	200	860	856.725 ± 1.5523	856.75 ± 1.945	856.825 ± 1.907	856.95 ± 1.81	856.7 ± 1.54
uf225-960	uf225-01	225	960	956.233 ± 2.34	956.5 ± 2.52	956.5 ± 2.53	955.967 ± 2.31	956.167 ± 2.56
uf250-1065	uf250-01	250	1065	1060.3 ± 1.588	1060.2 ± 1.42	1060.225 ± 1.405	1060.4 ± 1.56	1060.33 ± 1.57

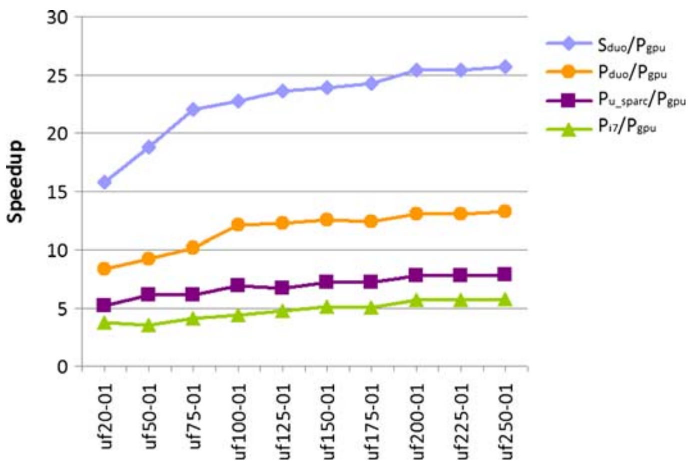

Fig. 11 Speedups with respect to implementations over different architectures

MAX-SAT problems. Parallel GAs (pGAs) are also in common use to reduce the total execution time.

Most of the pGAs developed in the last two decades were designed for clusters or MPPs. However, in this decade other architectures like GPUs are becoming increasingly famous for general purpose parallel processing. Frameworks like CUDA are used to take the maximum advantage of the low lying hardware using an industry standard programming language. Even though GAs are easy to parallelize, they do not give themselves easily to data parallel architectures like GPUs. For an efficient implementation, all the genetic operators and LS algorithms need to be redesigned with a considerable care.

We have recommended using a hybrid of cellular GA (cGA) and hill-climbing. cGA provides niching and its 2D structured population is ideal for implementation over an nVidia GPU using CUDA. Hill climbing in its simple form is not data parallel and hence is not suitable for implementation over GPU. We propose a data parallel approach to hill climbing with a feedback mechanism. Migration is another process applied to sub-population based pGAs which is not suitable for GPU. We propose a method called diffusion which suits thread based environments like GPUs. Selection, mutation, and recombination must also be adapted to the GPU architecture. Efficient use of data parallelism and memory is another key to success. In the results section we have shown a detailed analysis of different optimization techniques and the achievable speed-ups related with them.

We showed the contribution of this paper by solving a problem with many practical uses. This kind of approach can be used for other problems to achieve comparable speedups. For the time being the CPU is acting like a controller and the entire processing takes place on the GPU. Processing some part of the algorithm over CPU while the GPU is busy doing other processing would be a good area for future work. Moreover, further optimization of LS and implementation of other LS algorithms over data parallel architecture like GPU could be an interesting area of work. As another line of future work, the speedups achievable by GPU implementation can be used to design more complex algorithms that numerically outperform the existing algorithms.

Acknowledgments We would like to thank Dalila Boughachi for her help, regarding the use of genetic algorithms to solve MAX-SAT problem. We would also like to thank the anonymous users who replied to our queries regarding CUDA and GPGPU.

References

1. http://www.nvidia.com/cuda/
2. http://www.openmp.org/
3. nVidia CUDA Programming Guide 2.1 - CUDA 2.1 SDK Documentation (2008). http://www.nvidia.com/cuda/
4. E. Alba, B. Dorronsoro, *Cellular Genetic Algorithms, Operations Research/Computer Science Interfaces Series*, vol. 42 (Springer, 2008)
5. D. Boughaci, H. Drias, Solving weighted max-sat optimization problems using a taboo scatter search metaheuristic. in *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing* (ACM, New York, 2004), pp. 35–36
6. D. Boughaci, H. Drias, B. Benhamou, Solving max-sat problems using a memetic evolutionary meta-heuristic, in *Cybernetics and Intelligent Systems, 2004 IEEE Conference on Publication*, vol. 1. (2004), pp. 480–484
7. H. Drias, Scatter search with walk strategy for solving hard max-w-sat problems, in *Proceedings of IEA- AIE2001m Lecture Notes in Computer Science, LNAI-2070* (Springer, Budapest, 2001), p. 3544
8. K. Fok, T. Wong, M. Wong, Evolutionary computing on consumer graphics hardware. IEEE Int. Syst. **22**(2), 69–78 (2007)
9. J. Frank, A study of genetic algorithms to find approximate solutions to hard 3cnf problems, in *Golden West International Conference on Artificial Intelligence* (Kluwer Academic Publishers, 1994)
10. H. Hoos, T. Stutzle, *Satlib: An Online Resource for Research on Sat* (IOS Press, 2000), pp. 283–292. SATLIB is available online at http://www.satlib.org
11. S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimization by simulated annealing. Science, Number 4598, 13 May 1983 **220, 4598**, 671–680 (1983)
12. C. Li, Exploiting yet more the power of unit clause propagation to solve 3-sat problem, in *ECAI'96 Workshop on Advances in Propositional Deduction* (Budapest, Hungary, 1996), pp. 11–16
13. C. Li, Anbulagan: Heuristics based on unit propagation for satisfiability problems, in *IJCAI* (1), (1997), pp. 366–371
14. J. Li, X. Wang, R. He, Z. Chi, An efficient fine-grained parallel genetic algorithm based on gpu-accelerated, in *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops* (IEEE Computer Society, Washington, DC, 2007), pp. 855–862
15. E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, Nvidia tesla: a unified graphics and computing architecture. Micro. IEEE **28**(2), 39–55 (2008)
16. D. Martin, L. George, L. Donald, A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
17. M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
18. M. Matsumoto, T. Nishimura, Dynamic creation of pseudorandom number generators, in *Monte Carlo and Quasi-Monte Carlo Methods 1998* (Springer, 2000), pp. 56–69
19. B. Mazure, L. Sais, E. Crmoire, A tabu search for sat, in *Proceedings of AAAI* (1997)
20. V. Podlozhnyuk, Parallel mersenne twister. CUDA 2.1 SDK Documentation (2007)
21. B. Selman, H. Kautz, B. Cohen, Noise strategies for improving local search (1994)
22. J. Stratton, S. Stone, W. Hwu, Mcuda: an efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign (2008)
23. M. Tomassini, *Spatially Structured Evolutionary Algorithms* (Springer, Berlin, 2005)
24. Q. Yu, C. Chen, Z. Pan, *Parallel Genetic Algorithms on Programmable Graphics Hardware*, vol. 3612 (2005), pp. 1051–1059