NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver

Master Project Final Report by

Yandong Wang

Submitted to the faculty of

Rochester Institute of Technology

In partial fulfillment of the requirements for the degree of

Master of Computer Science

Project Committee:

Chair: Alan Kaminsky

Reader: Stanisław Radziszowski

Observer: James Heliotis

Contents

Abstract

The contribution of this project is that it incorporated the advantage of the application of stochastic local search and genetic algorithm to the SAT solvers and the superior parallel computing capability of CUDA GPU, designed a highly efficient CUDA architecture-based incomplete SAT solver that couples cellular genetic algorithm and random walk local search. The measurement results by testing Uniform Random-3-SAT SAT benchmarks present that this novel SAT solver is able to give out an efficient running time performance and ideal scalability.

Key Word: *SAT Solver, Parallel Computing, CUDA GPU, Genetic Algorithm and Local Search*

1 Introduction

The satisfiability problem is the first NP-complete problem which is seen as the fundamental of computing theory. Its exponential complexity has been challenging the most talented computer scientists for decades. Although $NP \neq P$ is still an unsolved problem, the emergence of number of sophisticated SAT solvers has leveraged the ability of solving SAT problem instances involving hundreds of thousands of variables and clauses. Since the advent of *Chaff* SAT solver [1] in year 2000, the significant performance improvement of SAT solvers unleashed their potential computation ability beyond their traditional application domain. So far, high-performance SAT solvers are widely used in real-world applications, such as electronic design automation, automatic test pattern generation, formal verification, and artificial intelligence. Besides these, SAT solvers have been extendedly applied with success on cryptographic problems that usually are based on the unproven hypothesis [2]. In spite of these phenomenal improvements, the industries'desire to meet the demand of larger industrial instances is still continuously pushing the computer scientists to pursue more efficient solutions.

Over the past decade, revolutionary breakthroughs and innovations in the computer hardware are again and again pushing forward the computation ability of SAT solvers and still keeping advancing the state of the art. With the rise of multiprocessing computing resources, the reality that designing parallelizable SAT solver algorithms will be the inevitable trend comes into focus. Numbers of parallel SAT solvers [3, 4] have been designed and implemented on parallel computing architectures. Some of them achieved unprecedented performance improvement. While, despite the promise of complete SAT solvers that they always give out the results, contemporary parallel complete SAT solvers are still mired in the problems like load-balance and low scalability. In addition, modern multiprocessing computation resources, such as cluster, hybrid computation network, are still expensive in terms of monetary cost. At this point, the emergence of CUDA GPU is changing the situation since so far it is a personally affordable massive parallel multiprocessing architecture. So we started considering an interesting question:"In order to overcome the disadvantages that are brought by implementing the complete SAT solvers on traditional parallel architectures, can CUDA GPU, this novel parallel computation architecture, be applied to design a parallel incomplete SAT solver and meanwhile achieve a satisfactory running time performance and scalability?" The test results of CUDA-based SAT solver designed in this project gave this question a very positive answer.

In the following, I am going to give the report of my design decisions while

developing this CUDA-based incomplete SAT solver. In order to enable this paper to be self-explained, in section 2, I present the background knowledge of the *satisfiability problem.* In section 3, I introduce the *CUDA programming*. In section 4, I describe all of the concrete design details of the new SAT solver. In section 5, the measurement and observation of the new SAT solver are presented. Future work and related research are presented in section 6 and section 7 respectively. Finally, the conclusion is in the section 8.

2 Satisfiability Problem and SAT Solvers

2.1 Satisfiability Problem

The SAT problem is the shorthand of *boolean satisfiability problem* which refers to the question that: "Given a boolean expression, determine if there exists an assignment of TRUE(1) or FALSE(0) to all boolean variables that make the entire expression to be TRUE." Equally important question is to determine whether or not no such assignment exists. Both of them are NP-complete problems [5]. In the first case, finding one assignment that enables the boolean expression to be TRUE is enough, if such assignment exists, we call this boolean expression is satisfiable, otherwise we call it is unsatisfiable which is proven in the second case that needs exhaustive search of all of the possible assignments.

Each SAT problem instance consists of boolean variables, "AND $\bigwedge$", "OR $\bigvee$", "NOT $\neg$" operators, and parenthesis. Besides these, there are some other terminologies associated with SAT problem. A "*literal*" is a boolean variable or its negative form. A "*clause*" is a disjunction of a set of literals concatenated only by "OR $\bigvee$" operator.

For example:

"a", "b", "$\neg$a" are literals. "(a $\bigvee$ b)", "($\neg$a $\bigvee$ b $\bigvee$ $\neg$c)", "a" are clauses.

A boolean expression formula is called *Conjunctive Normal Form*(CNF) expression if all of the literals are associated with one of the clauses, and all clauses are concatenated with each other by "AND $\bigwedge$" operators.

For example:

"($\neg$a $\bigvee$ b $\bigvee$ $\neg$c) $\bigwedge$ (a $\bigvee$ b) $\bigwedge$ c" is a boolean expression in CNF.

According to the rules of logical equivalence, each boolean expression formula

can be transformed into CNF form which sometimes is able to simplify the problem to some extent for exposing the underlying structure of the SAT problem, so that a couple of optimization strategies can be applied to reduce the size of the original problem. In addition, boolean expressions in CNF can be easily treated as input for SAT solvers. In this project, SAT problem inputs to the solver are all in the CNF form [1].

The boolean expression in CNF form with each clause contains at most k literals is called *k-SAT* problem. Of special interest are *2-SAT* problem, *3-SAT* problem and *MAX-SAT* problem that is to find an assignment that satisfies the maximum number of clauses in the problem. Like SAT problem, *3-SAT* and *MAX-SAT* are also NP-complete problems. But *2-SAT* is NL-complete problem [6] that is solvable in polynomial time. With regard to the *k-SAT* problem there is an interesting phenomenon. Random *k-SAT* problems exhibit a so-called "*Phase Transition Phenomenon*" [7], which is when there are exactly k literals in each clause, randomly choose the number of clauses c_k and the number of variables v_k, the probability of the satisfiability of the problem falls sharply from near 1 to near 0 as the ratio

$$r_k = \frac{c_k}{v_k} \tag{1}$$

passes some critical point called threshold, when k=3, the threshold value is about 4.25 (Figure 1). While it is much more complicated to find the threshold value once k is larger than 3. In Figure 1, When r_k is close to Y axis, the problem can be easily proved as satisfiable. Conversely, the problem can be easily proved as unsatisfiable when it is far from Y axis. The hardest instances appear at the region near the peak (when $r_k \approx 4.25$), during this region, enormous search space needs to be traversed until the solution is found. Based on this knowledge of the *k-SAT* problem, appropriate SAT problems can be selected as test cases[2].

[1]The input to the solver can be any text format file, the first line declares the number of variables and the number of clauses in the problem. Since second line, each line in the file is a clause of the SAT problem. Each clause consists of several integers which are the indices of the variables in the variable set. And the number of lines minus one is equal to the number of clauses.

[2]The test cases in this project are "Uniform Random *3*-SAT benchmarks" with the ratios r_k are around 4.25

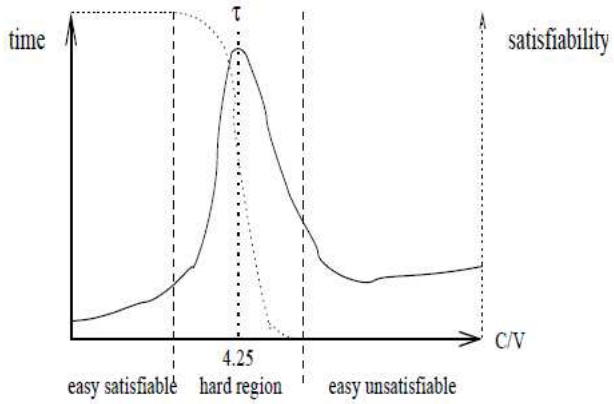

Figure 1. *3*-SAT Problem Phase Transition Phenomenon [7]

2.2 Satisfiability Problem Solver

Among all of the SAT solvers, two main categories of SAT solvers are widely studied by researchers, respectively there are *Complete SAT solvers* and *Incomplete SAT solvers*.

2.2.1 Complete Satisfiability Problem Solver

Complete SAT solver is the algorithm that aims at checking the satisfiability of the SAT problems. It guarantees to give the result of whether a SAT problem is satisfiable or unsatisfiable. Most of modern complete SAT solvers are based on the classical DPLL algorithm [8]. Even so, DPLL itself is still a highly-efficient procedure for SAT problems even under contemporary performance standards. The fundamental principles of DPLL algorithm are *backtracking* and *divide-and-conquer*. It firstly simplifies the problem by assigning some values to some variables, so if the rest smaller problem is satisfiable, then the entire formula is satisfiable, otherwise it goes back to assign the opposite values to the appropriate previously assigned variables, and keep doing this recursively until a solution is found or the entire search space is traversed.

During this process, DPLL actively calls two subroutines *Unit Propagation* and *Pure Literal Elimination* to enhance the efficiency of the algorithm, and recent researches are also eagerly looking for efficient approaches to improve these

two functions.

Unit Propagation: Some clauses only contain one literal, there is only one choice for the value of the corresponding variable, then these variables can be safely eliminated from the problem without affecting the search of the values of other variables. In addition, these eliminations may lead to the *deterministic cascades of unit clause* which is able to dramatically reduce the size of the original problem to avoid naive search or early detection of assignment confliction which is able to prove the unsatisfiability of the problem.

Pure Literal Elimination: If one variable occurs in the problem with only one form (positive or negative), then all of the clauses that contain this variable can be eliminated from the problem since the boolean value that makes the corresponding literal be true can make all of those clauses be true, and there is only one choice for this value. While *Pure Literal Elimination* is not used in DPLL as intensively as *Unit Propagation*, because finding all of the clauses contain single form variable is a computation intensive process, sometimes, it is not worthwhile.

Although modern DPLL-based algorithms have been phenomenally improved, their lack of parallelizability make them, if not unlikely, very complicated to be implemented on large multiprocessors parallel environment [9]. Recently, there are some other parallel complete SAT solver algorithms come forth, such as Join-and-Check based Solver [9], while most of them are still in the start-up phase.

2.2.2 Incomplete Satisfiability Problem Solver

Incomplete SAT solver is the algorithm that dedicated to finding the solution for the SAT problems during its running time. While the result is not guaranteed besides Incomplete SAT solvers are not able to prove the unsatisfiability of the SAT problem [3]. But its ability of quickly discovering the solution for certain kinds of pretty large satisfiable instances compensates its weakness to a great extent.

Incomplete SAT solvers are mostly based on the stochastic local search, and genetic algorithms, most of which are very suitable for parallel computing architecture. This grants the incomplete SAT solvers incomparable advantage of

[3]Because proving the unsatisfiability needs exhaustive traverse of the possible assignments, Incomplete SAT solvers only traverse as many assignments as it is able to during its running time.

Input 1: Randomly generated boolean assignment P
Input 2: Maximum Try Times M
while (P doesn't satisfies all clauses) AND (M is not reached) **do**
  C $\Leftarrow$ Select a random unsatisfied clause.
  V $\Leftarrow$ Select a random variable from C
  Flip the value of V
end while
if P satisfies all clauses **then**
  Return P
else
  No Solution is Found
end if

Figure 2. Pure(unbiased) Random Walk For SAT Problem [11]

parallelizability. In this project, the new CUDA-based SAT solver utilized this capability by implementing and improving the algorithm that based upon the previous research of parallel SAT solver that couples the cellular genetic algorithm and random walk strategy [10].

Random walk strategy [11] for SAT problem is one of the most efficient methods to search the solution for SAT problems by making use of the heuristic variable selection. It evolved from a *Pure (unbiased) random walk selection strategy* (Figure 2) to a *Biased heuristic search strategy* (Figure 3*a* abd 3*b*) in order to overcome the lack of knowledge about the objective SAT problem. While despite its performance improvement in the sequential execution, *Random walk strategy* alone is not suitable for parallel environment, since all the flip and search operations are executed on the same set of clauses. Each flip would affect each other unpredictably without guarantee to increase the number of satisfied clauses after operation. But there is one point in the random walk strategy that can be exploited to implement parallelization, since it needs a randomly generated boolean assignment as input to initialize the routine, if each random walk routine can get a different assignments as initial input, then they are able to search the solutions in different region of the search space simultaneously. The rest problems are how to give different assignments to discrete random walk routines at the same time and keep the diversity of the search space to avoid local minima.

Coupling with the ***Cellular genetic algorithm*** is the solution for the these problems. As standard genetic algorithms, Cellular genetic algorithm partitions the whole problem into smaller subproblems, each of which is responsible for the initialization, evaluation, reproduction, and evolution of its own population. In addition, it is designed by applying diffusion model which is an approach to implement genetic algorithm on parallel architecture. In diffusion model, each

```
Input 1: Randomly generated boolean assignment P
Input 2: Maximum Try Times M
while (P doesn't satisfies all clauses) AND (M is not reached) do
   C ⇐ Select a random unsatisfied clause.
   if There exists a variables V1 with break value = 0 then
      Flip V1
   else
      V ⇐ Select a random variable from C
      Flip the value of V
   end if
end while
if P satisfies all clauses then
   Return P
else
   No Solution is Found
end if
```

Figure 3*a*. Biased Random Walk For SAT Problem(version-1) [11]

```
Input 1: Randomly generated boolean assignment P
Input 2: Maximum Try Times M
Input 3: Probability ρ
while (P doesn't satisfies all clauses) AND (M is not reached) do
   C ⇐ Select a random unsatisfied clause.
   if There exists a variables V1 with break value[a] = 0 then
      Flip V1
   else
      if created Probability ≤ ρ then
         V ⇐ Select a random variable from C
         Flip the value of V
      else
         Flip the variable with the smallest break value
      end if
   end if
end while
if P satisfies all clauses then
   Return P
else
   No Solution is Found
end if
```

Figure 3*b*. Biased Random Walk For SAT Problem(version-2) [11]

[a]Break value is the number of clauses that become unsatisfied after flip

subpopulation is spawn from different individual assignment ancestors, each of which is organized in a discrete and independent cell of a low-dimensional grid (1-dimensional or 2-dimensional). And each individual only interacts with its directly connected neighbors to reproduce the new generations, von Neumann Neighborhood [12] and Moore Neighborhood [12] (Figure 4) are the two most common neighbor forms in cellular genetic algorithm. Each individual in the center cell only mates with one of the individuals that reside in the surrounding four or eight cells. The evolution process requires two individuals as parents, first of which is the individual in the cell itself, the other one is selected from neighbors by comparing their fitness [4]. After selection, the mutation and crossover operations are applied to parents individuals to produce new generation, which would replace the current parent individual in the cell no matter if the new generation has a better fitness or not to finish the evolution to order to keep the diversity. Above all, all of these operations can be performed by all of the individuals respectively at the same time, and each generation of assignments can be the inputs for the

[4]In the case of this project, the fitness is the number of satisfied clauses that a boolean assignment satisfies

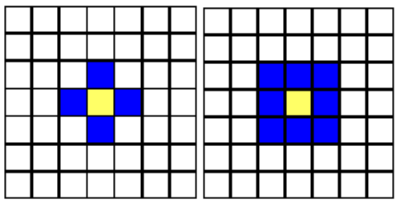

Figure 4. The left is von Neumann neighborhood. The right is Moore Neighborhood

above mentioned discrete random walk routines, so that parallelizability can be achieved.

The another critical problem is how to keep a steady diversity of the population so that a wider search space can be traversed. Population homogeneity is always a plague for standard genetic algorithms. Also, since both cellular genetic algorithm and random walk are using greedy approach, premature convergence could also be a potential poison for this combination. While previous researches [10, 14] have presented amount of measurement showed that the this combination of cellular genetic algorithm and random walk strategy is able to provide a healthy population diversity if it is implemented with a well-designed random number generator and the utilization of improved mutation and crossover operations [13], both of which have the crucial characteristics of bringing the population towards a better fitness and meanwhile introducing new individuals to maintain diversity.

3 CUDA GPU Progamming

GPU is a multi-threaded, many-core, massively parallel processor with tremendous computation power and high memory bandwidth. It is designed specifically for computing high parallel intensive-computation, like graphic float operation. The most obvious discrepancy (Figure 5) between CPU and GPU is that GPU concentrates on similar data processing rather than data caching, flow control which are always the focuses of the design of CPU.

In order to fully take advantage of this incomparable strength of GPU, recently, graphic card company NVIDIA is aggressively advertising its new general purposed computing architecture "*CUDA GPU*" which enables dramatic increases in computing performance by harnessing the computing power of GPU. In addtion, NVIDIA published CUDA SDK and programming interfaces to pro-

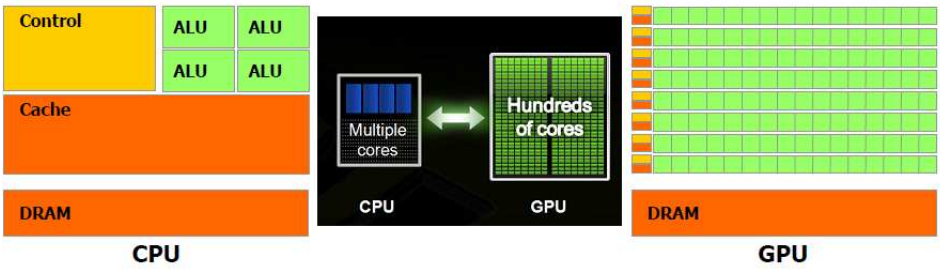

Figure 5. Difference between CPU and GPU [15]

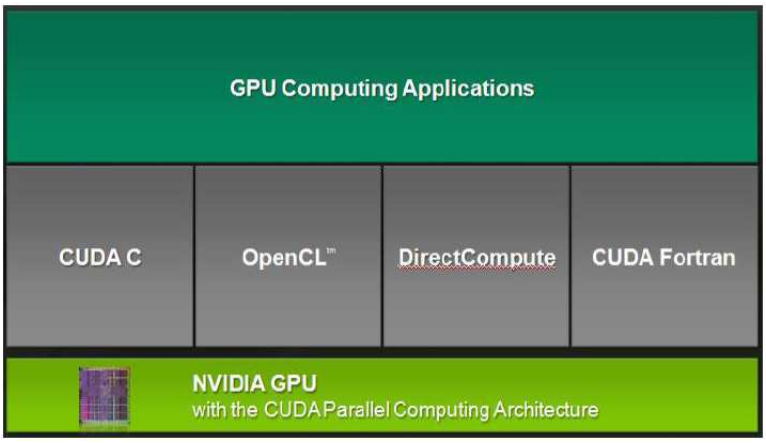

Figure 6. CUDA Programming Structure [15]

vide developers with a programmable platform on which high-level programming languages, such as C, FORTRAN, can be used to utilize the CUDA parallel computing engines in each NVIDIA CUDA-enabled GPU (Figure 6).

3.1 CUDA Programming Model

Each CUDA GPU has several multiprocessors (chips), each of which has several cores, all of those cores can run independent threads simultaneously. Each thread is organized into a one-dimensional or two-dimensional threads block. All of threads blocks are similarly organized into a one-dimension or two-dimenson blocks grid. Total number of threads during the running time is equal to the number of threads per block times the number of blocks per grid, while each thread

will not be running until the threads block containing this thread is assigned to the one of the available multiprocessors. Once this operation is done, all threads in one block are further divided into *k*-thread units called *warps*, the real size of *warp* is implementation specific and varies from implementation to implementation. Also, *warp* is not a CUDA language definition, which means there is no specific data structure describing the warp, instead it is just an implementation specification.

Usually the number of blocks needed in the program is associated with the size of the problem instead of the number of multiprocessors inside GPU, so that it can greatly exceeds the number of chips to be as large as 65535×65535. Conversely, although the number of threads in each block can exceed the number of cores inside each chip too, this number is always limited by the size of the memory shared by all threads in one block.

GPU is responsible for scheduling the blocks and threads, each block and thread can be scheduled on any available multiprocessor and core in any order, concurrently or sequentially. So during the running time, the execution order of blocks and threads are undefined, any attempt to write the program based on the execution order should be avoided. In addition, the execution of threads block are completely independent of each other, this feature enables computation to be transparent scalable, since GPUs with different number of multiprocessors can automatically assign each block to any available multiprocessor without worrying about the entanglement between different blocks.

CUDA threads are scheduled in a groups of warps, the threads within a warp are executed in a called *Single-Instruction Multiple Thread (SIMT)* manner, which means all threads in a warp execute same instructions at the same time, if there exists branch statements among the threads in a warp, GPU will serially execute them. For example, if there are 16 threads in a warp, and 8 threads execute the instructions in branch A, another 8 threads execute the instructions in branch B, GPU decides which branch is executed first, let say branch A, then all threads in B will not be executing until all threads in A finish, and then all threads in A wait until threads in B finish. So, too many conditional statements would dramatically hurt the parallel performance of entire program.

In CUDA C, each thread is defined in a called *kernel* function, which is declared by adding the *__global__* prefix before the function name. All *kernel* functions execute in parallel on a GPU device. The*__global__* function serves as a point of entry into the kernel and exposes a particular form of parallel computation called *data parallelism* that distributes a large task composed of many similar but independent pieces across a set of CUDA threads. In order to distinguish

each thread from each other, CUDA provides a built-in variable *threadIdx* to identify each thread in one threads block. So is threads block, which is accessible through the build-in variable *blockIdx* to retrieve the position of a block in a grid of blocks, and all threads in the same block share the same *blockIdx* variable. In addition, the dimension of threads block and blocks grid can be accessed respectively through *blockDim* and *gridDim* built-in variables. While, these variables are only accessible inside *__global__* or other device functions. Once the kernel is launched, the dimension will not change anymore. One thing need to be noticed is that *kernel* functions are only callable from host functions (regular C function) and are launched asynchronous, once the launch is finished the control immediately returns to the calling host functions before the kernels starting to executing. When calling the *kernel* function, the number of blocks per grid and the number of threads per block needs to be assigned to *__global__* function as parameters explicitly by using the CUDA defined *execution configuration syntax* $<<<\ >>>$, each such parameter can be either a *dim3* struct type if it is multi-dimension or a *int* type if it is 1-dimension, and they do not have to be compile-time constants.

A *__global__* function can only invoke device functions which are declared with *__device__* prefix that marks a function is callable from threads executing on the device. Compared to the *__global__* function, *__device__* functions behave more like regular C functions since the invocation does not need CUDA defined syntax. While the *__device__* function is not callable from host functions. The host function is the regular C function which is declared either without any prefix or with *__host__* prefix.

During the running time, CUDA allows programmers to coordinate all threads in the same block by using the barrier synchronization function *__synthreads*() within the device functions, unlike the synchronization mechanism in high-level programming language such as JAVA or C++, that prevents the other threads from accessing the synchronization block when one thread is in it, *__synthreads*() blocks threads from continuing until all threads in the block reach the same point. For this reason, all threads in the block should have approximate execution time otherwise faster threads have to wait slower threads to finish before proceeding which would abruptly reduce the performance. While there is no similar mechanism to synchronize all blocks under device programming environment, but CUDA provides *cudaThreadSynchronize()* function to coordinate all threads from all blocks under host programming environment.

```
__device__ int get_block_id();
__device__ int get_thread_id();

__global__ void kernel_function()
{
int block_id = get_block_id();
int thread_id = get_thread_id();
}

__device__ int get_block_id()
{
return blockIdx.y * gridDim.x + blockIdx.x;
}

__device__ int get_thread_id()
{
int inner_block_id = threadIdx.y × blockDim.x + threadIdx.x;
return get_block_id() × (blockDim.x × blockDim.y) + inner_block_id;
}

__host__ void host_function()
{
dim3 blocks(12,12);
dim3 threads(10,10);
kernel_function <<< blocks,threads >>> ();
}
```

Figure 7. CUDA Functions Demonstration

3.2 CUDA Memory Model

In CUDA GPU programming architecture, host and device have separate memory spaces. All threads in the device are only running on the device memory, the programmer is responsible for allocating the necessary device spaces, transferring the related data between host and device memory, and cleaning up all of the unnecessary device memory. CUDA run time system provides programmers with APIs to perform all of these activities.

During the execution, each thread may access data from different types of device memory (Figure 9). First of all, all threads in all blocks can read and write the *global memory* which is allocated and released by calling the *cudaMalloc(...)* and *cudaFree(...)* functions respectively. The global memory is the largest chunk of memory shared by all threads in the device, while its access has the highest latency and the lowest bandwidth among all types of device memory.

CUDA exposes a fast shared memory region (*16 KB*) in each multiprocessor that is shared by all threads in the same blocks. Shared memory on each chip is completely independent of each other, it has the fastest access, this capability can compensate the disadvantage of the small number of registers available for each

thread and be used as manageable data cache. Properly exhaustive usage of shared memory can significantly improve the performance, and any opportunity to replace the usage of global memory with shared memory should be exploited. Variables resides in the shared memory are declared with *__shared__* prefix. There are two ways to allocated shared memory: *static compile time allocation* and *dynamic running time allocation*. In the first case, the size of the variables needs to be declared explicitly during the compile time. In the second case, the size of the shared memory that will be used in each block is the third parameter in *execution configuration syntax<<<blocks, threads, shared memory size>>>* when calling the kernel function. Inside the kernel function, before using the shared variables, "extern *__shared__variable_name*" needs to be declared. GPU is responsible for the releasing of the shared memory.

Each thread has its own processor registers, whose available amount is relatively pretty small but has the fastest access, and local memory. Local memory is actually a conceptual memory space instead of a real memory block implementation. It resides in the global memory so it shares the global memory's high latency and low bandwidth attributes. The access to the local memory occurs only when there is no more available registers to each thread. The allocation and release operations of local memory are invisible to the programmer.

In addition to the above four kinds of read-and-write device memory, there is a kind of read-only constant memory. Constant memory is not writable for the device functions but host functions are still able to do the write operation on it. Since the usage of constant cache, its access is faster than the global memory, but still slower than shared memory. It is suitable for hosting large chunk of data which needs fast access and no modification request but is not able to fit into the shared memory. Constant variables are declared with *__constant__* prefix and can only be allocated at the compile time.

3.3 CUDA Code Compilation

CUDA C allows a mix of host functions and device functions. In order to compile the programs, NVIDIA's nvcc compiler is needed, which separates the device functions from regular C functions, then it compiles the device functions into *PTX* code which is CUDA instruction set and left the rest C functions to the regular C compiler. So compiling the CUDA code also needs regular C compiler.

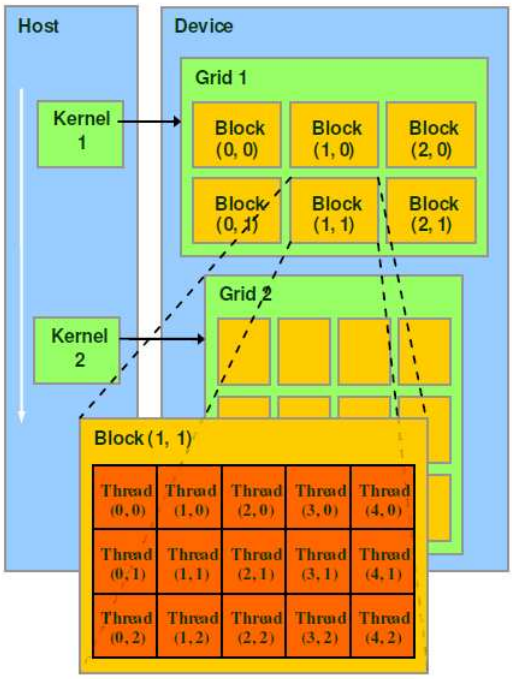

Figure 8. CUDA Threads Organization [15]

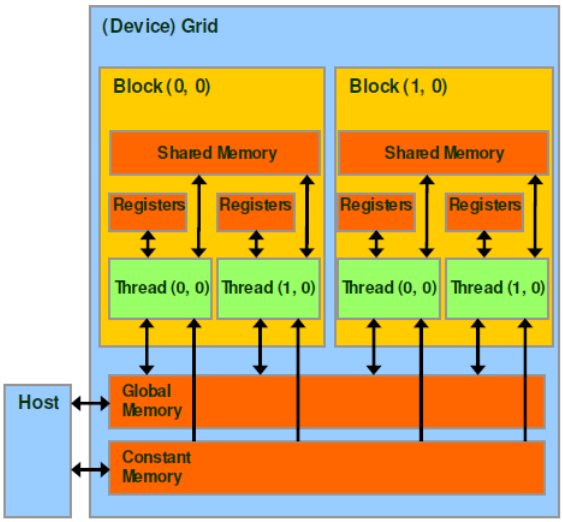

Figure 9. CUDA Memory Model [15]

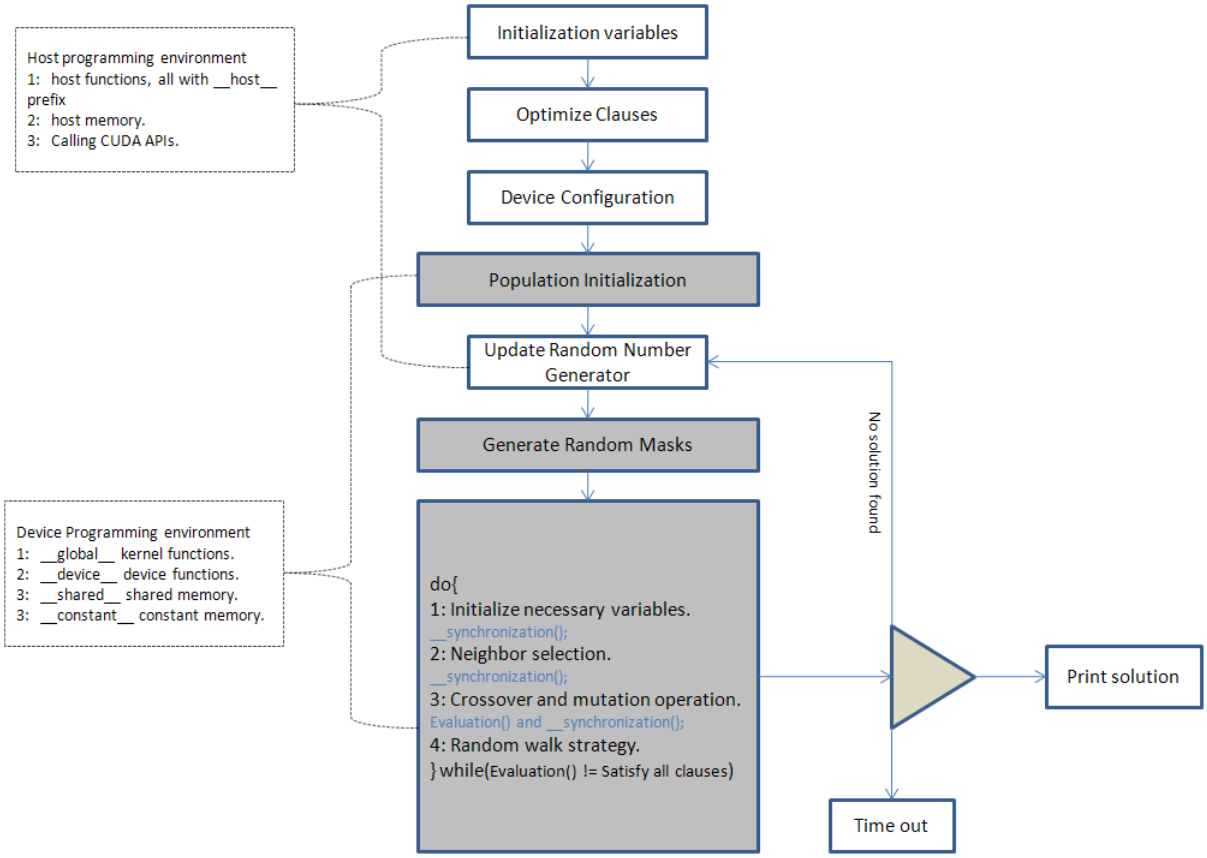

Figure 10. CUDA-based SAT Solver Profile

4 CUDA Architectured-based Parallel Incomplete SAT Solver Design

In this project, a new parallel incomplete SAT solver based on the cellular genetic algorithm coupling with random walk strategy by applying the CUDA-enabled GPU parallel computing environment was designed. The source code of entire project is a mix of host regular C functions and CUDA C device functions (Figure 10).

4.1 Device Memory Allocation

After reading the input from the initialization phase, all necessary data should be transferred into device memory which is done in the device configuration phase. While dumping all of the data into global memory is inefficient unless the size of data is huge. Among all of the inputs, the small type data, such as the number of variables in boolean expression, the number of clauses, should be assigned to the variables in the shared memory or at least constant memory, since these data are frequently visited by the device functions, such as in the conditional statements of loop routines. If the data is too large to fit into the shared memory, like clauses information, global memory is still the last choice, since clauses data is never modified during the running time, it can be assigned to the constant memory, the disadvantage of putting the clauses data into the constant memory is that the size of the constant array needs to be decided during the compile time.

Once the kernel is launched, each thread controls one individual truth assignment which is represented by a *char array* in which each bit represent one variable of the SAT problem. The number of truth assignments of one generation is equal to the number of all threads in all threads blocks. All of these assignments need to be firstly assigned to the device memory before kernel proceeding. The entire truth assignments data can be pretty huge considering the number variables in the SAT problem, and they are recurrently modified by each thread. What is more important is that each assignment need to be visible to its neighbor threads. For these reasons, global memory is the only choice. I allocated a matrix of *char array* over the global memory, each thread is in charge of one element of the matrix, and each block is working on one independent sub-matrix (Figure 11).

When each thread is processing its own truth assignment, such as doing the crossover, mutation, and evaluation operations, it is not necessary to access the truth assignment matrix which resides in global memory until the evolution replacement. So each thread has a chunk of shared memory as a truth assignment cache to accelerate the processing procedure, and all threads in all of the blocks that would be running on the same multiprocessor share the same chunk of shared memory. Since each multiprocess has a limitation of usable shared memory, if the number of variables is beyond the limitation, no shared memory cache is able to be assigned to each thread. Instead, each thread would have to use global memory as truth assignment cache to perform the operations.

In addition, each thread needs its own crossover mask and mutation mask to perform the following crossover and mutation operations, the length of each mask is equal to the number of variables of the SAT problem, all masks needs to be

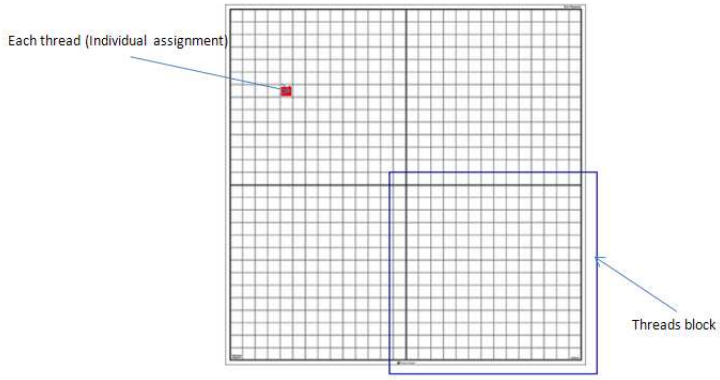

Figure 11. Truth Assignments Population Organization on Global Memory

regenerated after each round of solution search. In order to meet these two requirements, two arrays for masks were allocated over the global memory too.

4.2 Random Number Generator

Random number generator has one of the most important roles during this project. Whether it is designed properly at a great extent affects the diversity of the population during the evolution as described in section 2.2.2. The pseudo-random number generator in this project is based on the hash algorithm (Figure 12) defined in [17, 18]. In order to keep steady population diversity, only one random number generator is used in the entire program. While this hash algorithm itself is not a parallel algorithm, which means if one thread is using this generator, the other threads have to wait until it is released. This would make the population initialization and mutation-crossover masks generation process behave like sequential execution which sharply damages the performance. One of the parallel solutions to solve this problem is by using multiple random number generators at the same time, while the previous experiment showed a low quality of this solution, because using multiple random number generators naturally brings a high probability of overlapping the same search space region.

Sequence Splitting [19] is an approach to parallelize a sequential random number generator. It splits the random number sequence into several non-overlapping contiguous sections, each of which can be generated by different threads, and this

$x = 3935559000370003845 \times i + 2691343689449507681 (mod 2^{64})$
$x = x \oplus (x \gg 21)$
$x = x \oplus (x \ll 37)$
$x = x \oplus (x \ll 4)$
$x = 4768777513237032717 \times x (mod 2^{64})$
$x = x \oplus (x \ll 20)$
$x = x \oplus (x \gg 41)$
$x = x \oplus (x \ll 5)$
(The shift and arithmetic operations are all performed on unsigned 64-bit numbers.)
Figure 12. Hash Algorithm [18] for Generating the Random Numbers

process can be performed recursively. In this project, since the population initialization and masks generator both need continuous random numbers to achieve the purpose, *sequence splitting* is an appropriate solution, and experiment shows that it can give out a steady performance output.

While there is a tradeoff made here, the perfect random number sequence is generated by sequentially applying the hash algorithm over the previously generated random numbers, while this is an intensive sequential computation. So in order to accelerate the sequence jump speed, each random number sequence is generated by adding the length of the sequence to the previous sequence start point as input to the hash algorithm.

4.2.1 Random Initial Population Generation

After finishing device configuration, Initial truth assignments need to be assigned to every cell of the truth assignments matrix to enable kernel starting to search the solution. Since each cell is in the charge of the corresponding thread, it is the responsibility of each thread to initialize its own element of the matrix. The main objective of this phase is to minimize the probability of different threads generating the same truth assignment.

Each truth assignment is a *char array* in which each bit represents one variable of the SAT problem, so I generated a contiguous random number sequence, divided it into pieces by every *char* size bits, assigned each piece to one element of the *char array*. Because only one random number generator was being used in the entire system as described before, each thread got a copy of the same generator; applying sequence splitting needs each thread to jump to the appropriate start point to generate its own random numbers sequence for minimizing the overlapping population region. The jump length is:

$$jump\_length_1 = get\_thread\_id() \times \{\frac{number\,of\,variables N}{size\,of\,(char)} + (1\,or\,0)\} \tag{2}$$

4.2.2 Random Crossover and Mutation Masks Generation

Right Before starting to search the solution, each thread needs to generate its own crossover and mutation masks to perform the later operations. In addition to keeping the diversity, each mask also has to satisfy the probability of the occurrence of 1 in the mask is equal to the P which is a system parameter[5].

Different from generating the initial population which was doing bits copy operation, each bit in the mask should be generated discretely. So each bit of the mask needs one independent random number, both crossover and mutation masks need N random numbers respectively, N is the number of variables in the SAT problem. Same as in the population initialization process, each thread needs to jump to the right start point to generate its own random number sequence. The jump length is:

$$jump\_length_2 = get\_thread\_id() \times N \times 2 \tag{3}$$

4.3 Assignment Evaluation

Evaluation is to count how many clauses that one truth assignment satisfies, the result is the fitness of this assignment. Each variable in the SAT problem is given an index which also identifies its index position in the *char array*. When traversing the clauses data, the positive number means getting the bit value at $(|number| - 1)$ index. Conversely, negative number means getting the opposite bit value at $(|number| - 1)$ index.

In each *char array*, besides the bits representing the variables of SAT problem, there are also two extra chars used for storing the fitness value of this assignment. Each char respectively represents the high 8 bits and low 8 bits of the *short int* type, so that each assignment only needs to be evaluated once. The size of the *char array* is:

[5] P_c, P_m are the probabilities of occurrence of 1 for crossover mask and mutation mask respectively.

$$char\_array\_size = \frac{number of variables}{size of char} + (1 \, or \, 0) + 2 \tag{4}$$

4.4 Neighbor Selection

In this project, I am using von Neumann neighbor model to do the neighbor selection for reducing the number of conditional statements. Because all truth assignments are in the global memory, each thread can access the truth assignments of other threads. While in order to calculate the neighbor threads'id, conditional statements are unavoidable. This caused the problem that if there are more threads in one block, the longer the sequential execution will be. When the threads are inside the block boundary, the neighbor threads'id can be calculated by modifying the thread index. While if the threads are on the block boundary not only the threads index but also the block index need to be modified to get the neighbor threads'id (Figure 13).

Then each thread compares the neighbors'truth assignments, and picks the one who has the best fitness, groups with its own truth assignment to form a parents input for the following crossover and mutation operations.

4.5 Modified Crossover and Mutation Operations

In the original purposed algorithm, the author used a two-point crossover operation over the parents individuals to reproduce the new generation. It firstly selects two random positions h and l between 0 and N (The number of the variables), then swap both the left and right b closest bits to the position h and l of the parent assignments to generate two new assignments. While since this function needs actively doing the boundary checking which is working fine if under the regular CPU parallel environment, it would damage the performance under CUDA environment, since all threads in the warp can not achieve 100% parallelization. In order to overcome this problem, this project implemented a modified crossover operation combined with mutation operator.

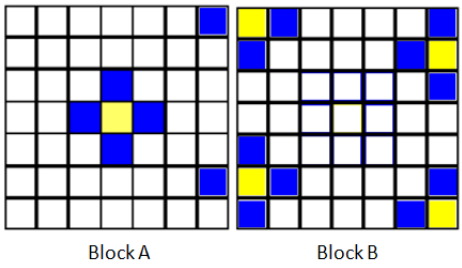

Figure 13. Neighbor Selection

$$child = (p_1 \wedge mask\_crossover) \oplus (p_2 \wedge \neg mask\_crossover)$$

$$child = child \oplus mask\_mutation$$

Figure 14. Modified Crossover, Mutation Combination Operation [13]

The modified crossover and mutation methods are based on the "AND $\wedge$" and "XOR $\oplus$" single-point operations [13] cooperating with the randomly generated crossover, mutation masks. The objective of crossover operation is to produce two new candidate assignments that will be identical to the parents assignments with probability (1 - P_c). The mutation method is used to improve the diversity of the population. While under device programming environment, producing two temporary children assignments needs compile-time knowledge of the number of variables to define *char array* if I want to use shared memory or local memory, either I could allocate necessary global memory during the initialization phase for this operation specifically, while after this operation, these memory would not be used again, it would be wasted and limit the number of threads in each block. In addition, repeated access to the global memory would not bring any performance improvement. In order to eliminate this problem, each crossover operation only produces one child assignment instead, so that the whole operation and following mutation operation can directly work on the individual assignment cache which is located in shared memory. By using this operations combination, conditional statements caused by boundary checking can be eliminated, each individual is able to execute the exact same instructions; Also, I can fully use C embedded bitwise operations to accelerate the performance. During this process each individual has its own random masks to perform the operations, so parallelization can be obtained.

4.6 Random Walk Strategy and Evolution

Random walk consumes most of the running time in each thread. It is by using the greedy strategy to find the best flips of the variables and bring the truth assignment towards the better fitness. During the process, it checks all of the variables back and forth M times[6] to test the fitness of the temporary new assignment after each flip of the value of the variable, if the flip can bring the fitness improvement, then replace the old assignment with the temporary one, since this operation is working directly on the individual assignment which is located at shared memory, the replacement operation does not need any other extra operations. But if the flip cannot bring any increase of the number of satisfied clauses, then value of the bit needs to be flipped back.

While because both random walk and previous neighbor selection apply the greedy strategy which would bring the premature convergence of the population search space, each assignment found after random walk always replaces the old assignment which is located in global memory, the motivation for this are that substitution of an assignment, even the fitness could get worse, helps maintain population diversity [10], and evolves with a sufficient grade of independence by exploring different portions of the search space.

5 CUDA SAT Solver Measurement and The Observation During the Testing

During the testing, the machine I used is Sun Microsystem Ultra 40 workstation with 1GHz dual-core AMD Opteron 2218 CPU, 8GB main memory. The GPU is *NVIDIA Tesla C870* which has 16 multiprocessors, each of which has 8 cores, and 1.5GB device memory, and its compute capability is 1.0.

The test case is *Uniform Random-3-SAT* problem set which is from the SATLib website. Since the CUDA-SAT solver in this project is an incomplete algorithm, all of the test cases used are satisfiable instances whose size are from the 20 variables, 91 clauses to 250 variables, 1065 clauses.

Again, this CUDA-based SAT solver aims at obtaining efficient running time and scalability, so the following measurement and analysis will be presented from these two aspects.

[6] M is the maximum flip time, configured as a system parameter

Problem Size (variable, clauses)	Minimum Running Time	Maximum Running Time	Running Time Difference
(20/91)	0.03	0.06	0.03
(50/218)	0.19	0.2	0.01
(75/325)	0.38	0.4	0.02
(100/430)	0.67	4.55	3.88
(125/ 538)	1.03	6.06	5.03
(150/645)	1.48	68.5	67.02
(175/753)	1.99	140.31	138.32
(200/860)	2.61	925.41	922.8
(250/1065)	4.04	546.53	542.49

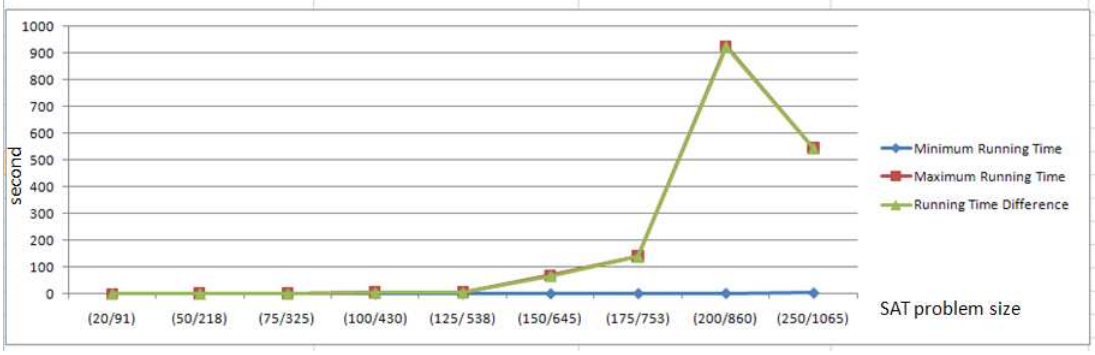

Figure 15. Running Time Difference of The SAT problems at the Same Size.

5.1 Running Time Measurement and Analysis

Testing the running time is to establish a basic knowledge of what size of SAT problems we can expect this solver to find the solution by using the existing computing resource. So the objective of testing the running time is to know the performance of this SAT solver for the specific size of SAT problems. During the testing, I used all 16 multiprocessors of the GPU, each multiprocessor contains 10×10 threads.

Because the solver uses only one random number generator during the entire process to generate the initial generation and all of the crossover, mutation masks, the speed of finding the solution at a great extent depends on the initial seed parameter. The observation shows that by using different seeds as input, the difference of the speed of finding the solution for the same problem can be very huge. In order to diminish this disturbance from seed parameter during testing the running time, each instance was tested 3 times by using 3 different seeds as input parameters, and the final running time result for this instance is the average of those 3 running time results.

Another unstable factor is that the hardness of finding the solution for different instances at the same size is unpredictable, I found the running time variation among small size SAT problems is much less than that among the large problems(Figure 15). So, for each size of SAT problems, I tested 10 different instances, so that the final running time of the specific size problem is the average of the running times of all those instances. The average running times for different size SAT problems are presented in Figure 16_a and 16_b, and the rate of increase of average running times are presented in Figure 16b. For all test cases, the solver found the solution before the time-out.

In Figure 16_a, we can see that for small SAT problems (less than 175 variables), the solver can give a perfect low and stable running time. In Figure 16_b, we can see a stable and linear average running time increase. From 16_c, we can see that the rate of increase of average running time is kept at a low level (About $2.5\times$ increase). While there are some fluctuation happened when the problem size increases. After thinking, I ascribe this hop to the following reason:

- Since the $r_k = \frac{c_k}{v_k}$ of all cases: ≈ 4.25, there is a high probability of meeting the hardest problems in the sets which need large traverse of the population until finding the solution . If the seed is not selected appropriately, it may take the solver to a search space which is far from the space in which the solution exists. Adding that the problem size has increased exponentially, evolution process from one search space to the other would take enormous times so that it would significantly increase the average running time for this size problem even the solver can find the solutions for other instances in pretty short running time. Shown in Figure 17.

Compare to the rate of increase of the problem size which is 2^{25}, increase shown in the Figure 16_b is acceptable. and from the trend of the rate of increase of average running time, we can establish a brief idea of how much time it may take to solve the larger problems. And even there are some fluctuations of running time and rate because of some hard instances which leverage the average time dramatically, the solver still presents a stable and low rate of increase of running time for all test cases in general. And it is able to find solution for most instances in pretty short period.

Average Running Time for different Size Random 3 SAT, Population Size 1600	
Problem Size (variable,clauses) \ threads	Grid.x = 16, Grid.y = 1, block.x = 10, block.y = 10
(20/91)	0.05
(50/218)	0.19
(75/325)	0.39
(100/430)	1.02
(125/ 538)	1.68
(150/645)	12
(175/753)	16.1
(200/860)	90.54
(250/1065)	111.51

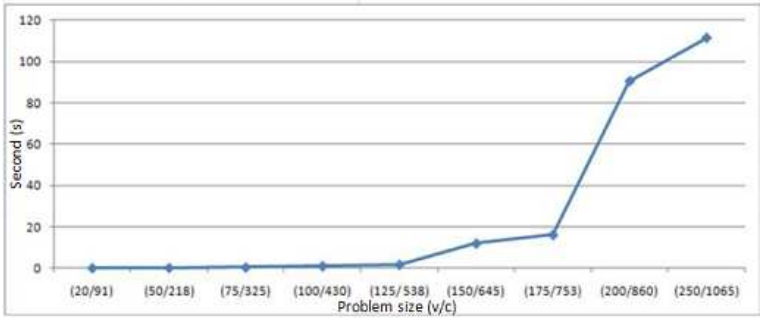

Figure 16_a. Average Running Time of Different Size of SAT problems.

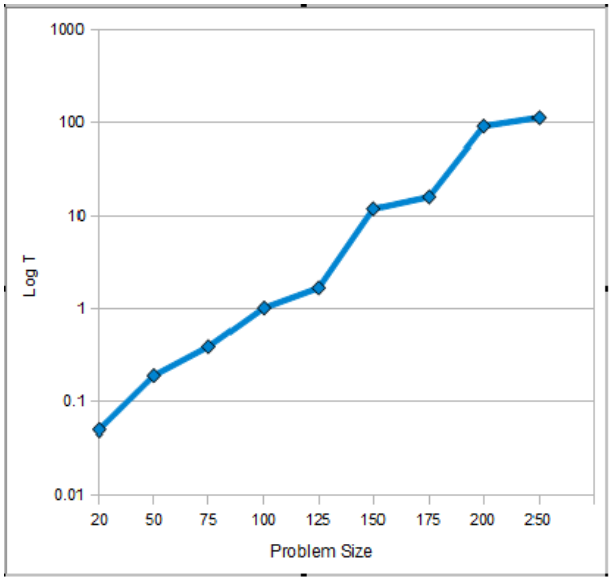

Figure 16_b. Average Running Time of Different Size of SAT problems (Log Based).

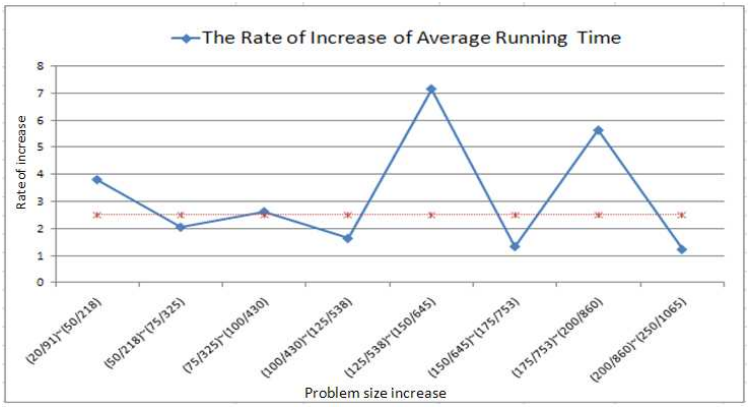

Figure 16_c. The Rate of Increase of Average Running Time of Different Size of SAT problems.

Problem Name	Second (s)		Problem Name	Second (s)
uf175-019	57.83		uf200-01	7.81
uf175-034	57.31		uf200-02	7.86
uf175-048	7.26		uf200-081	2.64
uf175-049	4		uf200-077	12.98
uf175-057	3.33		uf200-058	2.63
uf175-063	4.67		uf200-085	2.63
uf175-077	19.93		uf200-094	571.93
uf175-026	2.66		uf200-05	5.23
uf175-090	1.99		uf200-089	289.04
uf175-099	2.02		uf200-069	2.61

Figure 17. All Running Time for SAT problems with 175 variables, 753 clauses and 200 variables, 860 clauses

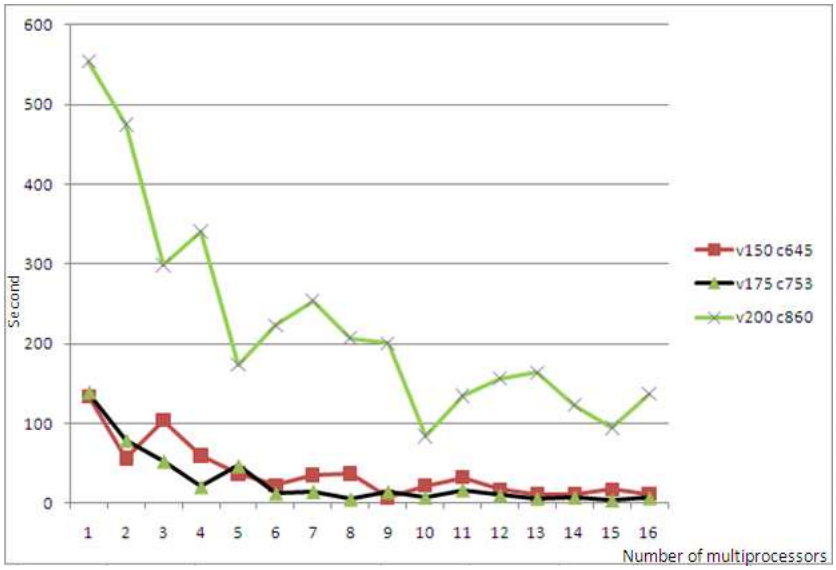

Figure 18. Naively Increase the Number of Multiprocessors

5.2 Scalability Measurement and Analysis

By testing the scalability of the solver, the knowledge of what kind of performance improvement this solver is able to get if it is applied on a stronger CUDA-based GPU with more multiprocessors can be constructed. I used different strategies to test the scalability. And the performance presents a very positive result.

Because the number of threads in each block is limited by the shared memory, there is a ceiling for this number which is 340 threads for each block.

The first strategy I used to test the scalability is:"***naively increase the number of multiprocessors while fixing the number of threads each multiprocessor holds***". The result is shown in the Figure 18.

In this is diagram, what we see is a non-linear leapfrogging falling instead of a linear speed up. Since when I increased the number of multiprocessors, it also triggered the alternation of several other factors.

- First, when I changed the number of multiprocessors, the entire population size, which is one of the main factors that affect the performance of cellular genetic algorithm, was also altered. Although the larger the population size is, the better performance could happen. This is not always the case unless the population increases significantly. If a small population get a right

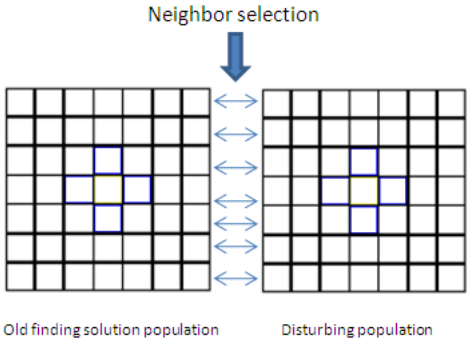

Figure 19. Neigbhor disturbance

seed to initialize its generation, it is more likely to find the solution in short period. While if increase the size of this population, because of the neighbor selection, the knowledge of population from other regions would infuse into the previous small region, which may lead the small population toward a complete different search space region. That is why large population does not definitely bring a faster performance. Shown in Figure 19.

- Once I significantly increase the number of multiprocessors, large diversity brought by the large population size increases the possibility of finding the solution. Conversely, that is why when the population size is small adding if the seed is not properly selected; it would take significant time to emigrate from one searching space to the other.
- Another observation is that once the number of multiprocessors passes some threshold, simply increasing the number would barely bring any performance improvement. This is because that when the only solution is contained in one part of the search space region; the entire running time depends more on how long it takes that region to find the solution than on the other population regions, unless the other regions are able to find another solution more quickly.

So from this case, since "The Population size = The Number of blocks $\times$ The Number of threads in each block". By using more multiprocessors, it is likely to improve the performance. While, in order to prove this hypothesis more deeply, in the second testing strategy, I tested 5 different SAT problems (250 variables,

Average Scalability of different SAT problems at same size	
Threads(block, threads)\SAT Problem	250 variables, 1065 clauses
[(8,1), (20,10)]	209.26
[(10,1),(16,10)]	138.18
[(16,1),(10,10)]	101.266

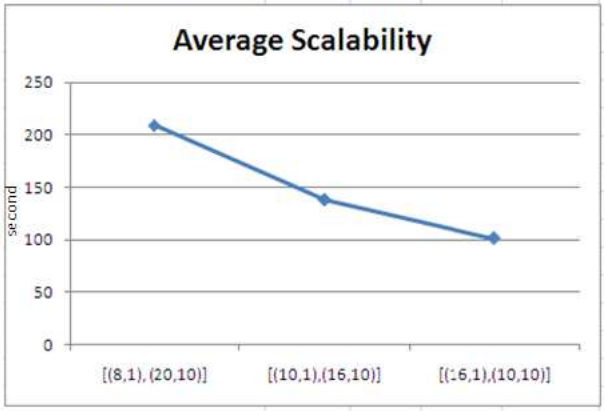

Figure 20. Average Scalability of Different SAT Problems at the Same Size

1065 clauses) at the same size while keeping the same population size. In each case, The Number of blocks $\times$ The Number of threads in each block = 1600. In addition, I organized all threads into a 160×10 2-dimension grid of threads to diminish the possibility that different threads organization may bring the different performance output.

Because of the variation of the hardness of each case, here I present the diagram (Figure 20) of average scalability of the 5 test cases. In this diagram, we can see an ideal scalability, especially when the number of used multiprocessors increases from 8 to 10, there is a super linear scalability increase. The reason is that in order to keep the same population size, the less the number of blocks is, more threads need to be in each block. While this would affect the performance significantly, because if there are more threads in one block, then there are more warps inside, if there exists conditional statement in kernel functions, this would lead to a longer sequential execution than that of if less threads reside in each block.

While the observation shows that the average scalability is basically decided by the scalability of finding solutions for hard instances, since it takes much more running time than that for simple instances, so even in some cases using less mul-

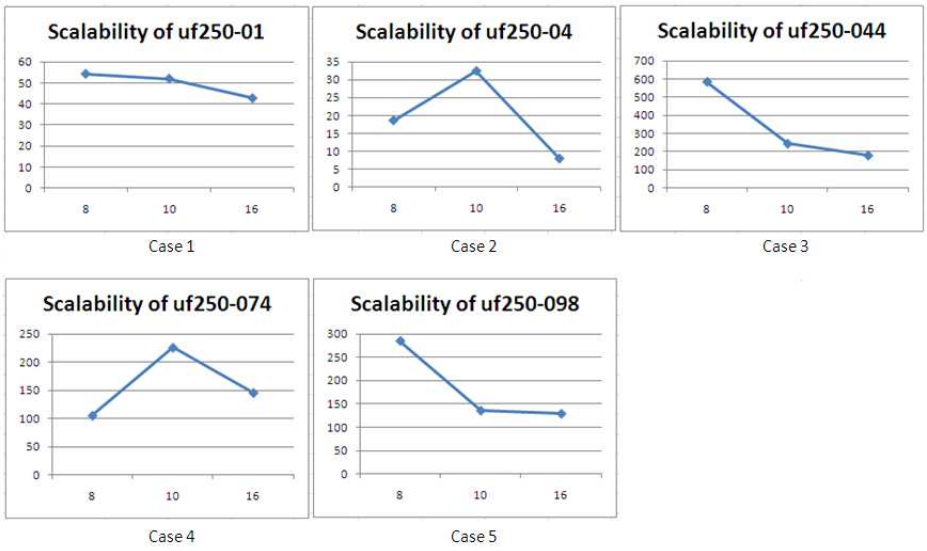

Figure 21. Scalability of Individual SAT Problem

tiprocessors can discover the solution faster than using more multiprocessors, it is compensated because of its less weigh when doing the average. Here I present the individual cases at Figure 21:

- In the case 1, we see a very mild scalability, because this is a relatively simple instance, it may contains several solutions spread around different search space regions.
- In the case 2, from the running time, we can also consider this as another relative simple instance, while the interesting point is that increasing the number of multiprocessors leads to a high running time. This is because even the population size and threads organization are same, each independent thread has different thread id when the organization of blocks is different (See Figure 7), then it will give a different permutation of the same initial generation (Figure 22). Then this will cause the crossover and mutation operations to generate a different children generation which may more quickly lead to the search space region where exists the solution.
- In the case 3, this is a relatively hard instance; it is more similar to the average scalability because all of the three running times are much more

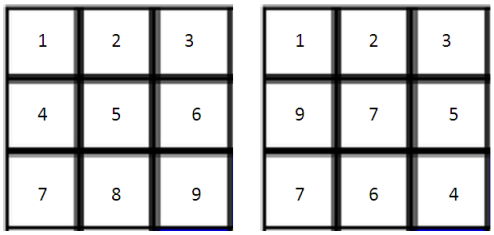

Figure 22. Same Population and Threads Ogranization, Different Permutation of Assignment

than those of the previous two cases, so even in the second case the running time of using 8 multiprocessors is less than using 10 multiprocessors, this affect is compensated by the running times in the hard instance. By using more multiprocessors, the population can emigrate from one search space region to the other more quickly, in addition, less threads in each block reduce the potential sequential time. Similar situation also happened in the case 5.

- In the case 4, it is more obviously affected by different permutation of initial generation.

From this test, the solver presents a good scalability for relatively hard instances, and poor scalability for relatively simple instances, While the short running time of finding the solution for simple cases can compensate this disadvantage.

6 Future Work

Although in this project the new CUDA-based SAT solver presents an efficient performance, there are still some points that can be improved.

1. Deciding a right seed for the solver is still an unsolved problem, its impact on the performance of each instance is too significant to ignore. The potential solution is to iterate a set of seeds, and set a time limitation for each iteration, if the solution can be found before the time-out, this seed can be seen as a proper solution for this instance.

2. There are too many *if* conditional statements for boundary checking in neighbor selection which to some extent hurt the performance. Although

there are some researches propose to use binary selection, which is a greedy strategy to select the neighbor, it still needs *if* branch. Another potential solution is to generate a boundary of dummy truth assignment elements in the matrix, while this needs a lot of extra device memory, in addition, the fitness value of dummy truth assignments will affect the evolution of the surrounding truth assignments toward the better fitness.

3. Currently, in order to accelerate the running speed, all the evaluation, crossover, mutation, and random walk operations are running in the shared memory, while there are only *16KB* shared memory for each multiprocessors, so the number of blocks resides on each multiprocessor and the number of threads in each block are limited by this number. For huge SAT problems, above operation need global memory as cache.

4. As mentioned in section 4.2 memory allocation, all clauses information are located in constant memory, while using constant memory needs compile-time knowledge of the size of the SAT problem, currently this number is set to 5000, which is relatively small for large SAT problem.

5. In this project, *Unit Propagation* and *Pure Literal Elimination* were not used actively, which may enhance the efficiency for large structured SAT problems.

6. The measurement showed that this CUDA-based SAT Solver is good at solving random *3-SAT* problems. While there is no systematic test on the structured SAT problems yet. This can also be the future work to improve this solver.

7 Related Research

In the [7], the author presented a detailed description of complete SAT solver algorithm and enumerated a collection of recent well-studied parallel complete SAT solvers. By the same auther, a novel complete SAT solver that is not based on DPLL algorithm is proposed in [9].

In the [20], the authors implemented a GPU-based *Survey Propagation* algorithm, which is also an incomplete SAT solver, by using NVIDIA *Cg* language, C++, and OpenGL under NVIDIA GTX 7900 GPU. The testing results presented

a very impressive performance for *3-SAT* hard instances. But unlike programming under CUDA environment, Under *Cg* programming environment, programmer is working on a high-level programming platform, device takes charge of the memory allocation, data transfer between memory spaces, and threads schedule.

In the [21], the researchers applied NVIDIA GeForce3 GPU computing power, built a general-purposed computing framework also based on *Cg* language. *3-SAT* problems were used as test cases proved that computing power of GPU can be applied not only to the graphic rendering computation but also to the high-performance required computation.

In the [22], there are thousands of technology reports about how people are using CUDA to improve the technology. As a novel high-performance computing platform, from enhancing the graphic rendering performance to solving the basic mathematic, biology problem, CUDA is filtering into every corners of modern technologies. Also in this website, there is a CUDA programmer community in which people together discuss the future and explore the potential of CUDA.

8 Conclusion

This project demonstrated that CUDA-enabled GPU can be an ideal parallel computing platform for the parallel incomplete SAT solvers that focus more on the similar data processing and intensive-computing. But the difficulties of designing complicate data structure under the CUDA GPU environment make it hard to optimize the process. In addition, there are too much restriction on the application of conditional statements which are the fundamental of programming. Although the measurement presents a good sign of this specific CUDA-based incomplete SAT solver, different design approaches are still needed to apply CUDA architecture to other kinds of parallel SAT solvers.

9 Acknowledgement

Here I want to say thank you to my advisor Professor Alan Kaminsky for his generous advice and suggestions during my project, and to my family, without

them, I would never finish this work.

References

[1] M.W.Moskewicz, C.F.Madigan, Y.Zhao, L.Zhang, S.Malik "Chaff: Engineering an Efficient SAT Solver" in Proc.of the Design Automation Conference, pages: 530-535, Year 2001.

[2] Mate Soos, Karsten Nohl and Claude Castelluccia "Extending SAT Solvers to Cryptographic Problems" In Theory and Applications of Satisfiability Testing - SAT 2009, pages: 244-257, Year 2009.

[3] Youssef Hamadi, Lakhdar Sais "ManySAT: a parallel SAT solver" Journal on Satisfiability, Boolean Modeling and Computation (JSAT), Year 2009.

[4] W.Chrabakh and R.Wolski. "GrADSAT: A parallel sat solver for the grid." Technical report, UCSB CS TR N. 2003-05, Year 2003.

[5] Cook, Stephen "The complexity of theorem proving procedures" Proceedings of the Third Annual ACM Symposium on Theory of Computing. pages: 151-158. Year 1971.

[6] Papadimitriou, C., Computational Complexity. 1994. Addison–Wesley.

[7] D.Singer."Parallel resolution of the satisfiability problem: a survey." In E.Talbi,editor. Parallel Combinatorial Optimization. John Wiley and Sons, pages: 123-147, Year 2006.

[8] Davis, Martin, Putnam, Hillary "A Computing Procedure for Quantification Theory" In Journal of the ACM 7. pages: 201-215, Year 1960.

[9] D.Singer, and A.Monnet. "JaCk-SAT: A New Parallel Scheme to Solve the Satisfiability Problem (SAT) based on Join-and-Check." In Proceedings 6th. Int. Conf. on Parallel Processing and Applied Mathematics, PPAM 2007, Gdansk, Poland, Springer Verlag LNCS 4967, pages: 249-258, Year 2008.

[10] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano "Parallel Hybrid Method for SAT That Couples Genetic Algorithms and Local Search" In IEEE transaction on evolutionary computation, VOL.5, NO.4, Year 2001.

[11] Wei Wei and Bart Selman "Accelerating Random Walks" In Principles and Practice of Constraint Programming, pages: 61-67, Year 2002

[12] Weisstein, Eric W., "von Neumann Neighborhood" from MathWorld.

[13] Lance Chambers "Practical Handbook of Genetic Algorithms: Complex coding systems" pages: 415-421, Year 2001

[14] A.Schoneveld, J.F.de Ronde, P.M.A.Sloot, and J. A. Kaandorp "A parallel cellular genetic algorithm used in finite element simulation " In Parallel Problem Solving from Nature Ů PPSN IV pages: 533-542, Year 2006

[15] NVIDIA CUDA Programming Guide. version 3
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/*NVIDIA_CUDA_Prograr*
Last accessd date: MAY 10 2010

[16] Alan Kaminsky, http://www.cs.rit.edu/ ark/spring2009/736/ 4005-736 Parallel Computing II, GPU Computing: Introduction to GPGPU and CUDA. 2009

[17] W.Press et al., Numerical Recipes: The Art of Scientific Computing, Third Edition (Cambridge University Press, 2007), page 352.

[18] Alan Kaminsky, http://www.cs.rit.edu/ ark/pj/doc/index.html
Parallel Java Library Documentation.

[19] Alan Kaminsky "Building Parallel Programs SMPs, Clusters, and Java" Cengage Course Technology, 2010, ISBN 1-4239-0198-3

[20] P.Manolios and Y.Zhang, ŤImplementing Survey Propagation on Graphics Processing UnitsŤ in International Conference on Theory and Applications of Satisfiability Testing (SAT 2006). Seattle, WA: Springer, pages: 311-324, Year: Aug 2006.

[21] Chris J. Thompson, Sahngyun Hahn and Mark Oskin. Using Modern Graphics Architectures for General Purpose Computing: A Framework and Analysis In Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. ISBN ISSN:1072-4451, 0-7695-1859-1, Pages: 306-317, Year 2002.

[22] NVidia CUDA for research http://www.nvidia.com/object/cuda_research.html

A Submission Materials

Submission materials include:

- C source code package: cuda_satsolver.cu, host_functions.h, kernel_functions.h, Random.h, Utility.h. All of the source codes are commented.

  - cuda_satsolver.cu includes the main function which is the entry for the program.
  - host_functions.h includes all of the functions run on the host machine.
  - kernel_functions.h includes all of the functions run on the CUDA-enabled GPU device.
  - Random.h is the source code of device hash function I get from Professor Kaminsky's website "http://www.cs.rit.edu/ ark/spring2009/736/gpgpu/attack01.shtml".
  - Utility.h includes all of the basic definition of data type and constant variables.

- All test cases: All of the *Uniform Random-3-SAT* problems I used as test cases during my testing phase.
- Measurement Record. In the Appendix C of this report.
- Master Project Paper Report.

B How to Run the Program

Configuration needs to do in order to run the program:

- An appropriate GPU drive needs to be installed based on the operation system, and GPU version. Those information can be found in the website: "http://developer.download.nvidia.com/compute/cuda/3_0/docs/GettingStartedLinux.pdf"
- Install the CUDA SDK and Toolkit.
- Make sure there is a C compiler in the machine. For Windows Operation System there is some information about how to use Visual Studio 2008 to

do the CUDA programming on the website:
"http://developer.nvidia.com/object/cuda_3_0_downloads.html".
In this project, I used Linux gcc compiler as C language compiler.

- Compile the Program using CUDA nvcc command: make sure the library path is set properly.
  Configuration under Linux :
  export PATH=/usr/local/cuda/bin:$PATH
  if the Operation System is 32-bit:
  export LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH
  if the Operation System is 64-bit:
  export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH

- Input Command mv a.out satsolver

- Run the program by using:
  ./satsolver SAT_FILE_NAME MAXIMUM_TRIES BLOCK_X BLOCK_Y THREAD_X THREAD_Y SEED
  MAXIMUM_TRIES = Maximum try times
  BLOCK_X = The number of blocks on the x axis of grid
  BLOCK_Y = The number of blocks on the y axis of grid
  THREAD_X = The number of threads on the x axis of block
  THREAD_Y = The number of threads on the y axis of block
  SEED = The seed used to initialize random number generator

C Testing Result of Running Time and Scalability

20 variables, 91 clauses,1600 population							
uf20-01							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.06	
5000	16	1	10	10	300	0.06	0.056667
uf20-02							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.06	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.05	0.053333
uf20-010							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.05	0.05
uf20-015							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.05	0.05
uf20-020							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.06	0.053333
uf20-025							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.06	0.053333
uf20-030							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.05	0.05
uf20-35							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.05	
5000	16	1	10	10	300	0.03	0.043333
uf20-40							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.06	
5000	16	1	10	10	300	0.06	0.056667
uf20-0222							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.05	
5000	16	1	10	10	200	0.06	
5000	16	1	10	10	300	0.06	0.056667

50 variables, 218 clauses, 1600 population size							
uf50-01							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0133							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.2	
5000	16	1	10	10	200	0.2	
5000	16	1	10	10	300	0.19	0.196667
uf50-0178							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0248							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0298							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.2	0.193333
uf50-0579							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0778							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.2	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.193333
uf50-086							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0986							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19
uf50-0996							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.19	
5000	16	1	10	10	200	0.19	
5000	16	1	10	10	300	0.19	0.19

Running Time: 20 variables, 91 clauses,1600 population and 50 variables, 218 clauses, 1600 population

75 variables, 325 clauses, 1600 population							
uf75-01							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	0.4	
5000	16	1	10	10	200	0.4	
5000	16	1	10	10	300	0.4	0.4
uf75-027							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.39	
5000	16	1	10	10	200	0.39	
5000	16	1	10	10	300	0.39	0.39
uf75-031							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.39	
5000	16	1	10	10	200	0.4	
5000	16	1	10	10	300	0.39	0.393333
uf75-056							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.4	
5000	16	1	10	10	200	0.39	
5000	16	1	10	10	300	0.39	0.393333
uf75-057							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.39	
5000	16	1	10	10	200	0.4	
5000	16	1	10	10	300	0.4	0.396667
uf75-063							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.38	
5000	16	1	10	10	200	0.38	
5000	16	1	10	10	300	0.39	0.383333
uf75-071							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.39	
5000	16	1	10	10	200	0.38	
5000	16	1	10	10	300	0.38	0.383333
uf75-078							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.4	
5000	16	1	10	10	200	0.4	
5000	16	1	10	10	300	0.4	0.4
uf75-09							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.38	
5000	16	1	10	10	200	0.39	
5000	16	1	10	10	300	0.39	0.386667
uf75-096							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.39	
5000	16	1	10	10	200	0.39	
5000	16	1	10	10	300	0.39	0.39

100 variables, 430 clauses, 1600 population							
uf100-0116							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	0.68	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	0.67	0.676667
uf100-0264							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.68	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	0.68	0.68
uf100-0300							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.67	
5000	16	1	10	10	200	0.67	
5000	16	1	10	10	300	0.67	0.67
uf100-0550							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.67	
5000	16	1	10	10	200	0.67	
5000	16	1	10	10	300	0.67	0.67
uf100-0695							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.67	
5000	16	1	10	10	200	0.67	
5000	16	1	10	10	300	0.67	0.67
uf100-072							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.68	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	2.62	1.326667
uf100-0748							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.67	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	0.67	0.673333
uf100-0895							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	0.67	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	0.67	0.673333
uf100-09							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.32	
5000	16	1	10	10	200	0.68	
5000	16	1	10	10	300	0.67	0.89
uf100-0989							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	3.26	
5000	16	1	10	10	200	4.55	
5000	16	1	10	10	300	1.97	3.26

Running Time: 75 variables, 325 clauses, 1600 population and 100 variables, 430 clauses, 1600 population

125 variables, 538 clauses, 1600 population								150 variables, 645 clauses, 1600 population							
uf125-010								uf150-023							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)	Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	6.06		5000	16	1	10	10	100	x	
5000	16	1	10	10	200	3.05		5000	16	1	10	10	200	13.1	
5000	16	1	10	10	300	3.04	4.05	5000	16	1	10	10	300	33.45	23.275
uf125-022								uf150-03							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.04		5000	16	1	10	10	100	42.45	
5000	16	1	10	10	200	1.04		5000	16	1	10	10	200	1.5	
5000	16	1	10	10	300	1.04	1.04	5000	16	1	10	10	300	20.51	21.48667
uf125-039								uf150-037							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.04		5000	16	1	10	10	100	2.93	
5000	16	1	10	10	200	2.05		5000	16	1	10	10	200	2.93	
5000	16	1	10	10	300	1.04	1.376667	5000	16	1	10	10	300	1.48	2.446667
uf125-047								uf150-044							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.03		5000	16	1	10	10	100	59.34	
5000	16	1	10	10	200	1.04		5000	16	1	10	10	200	4.37	
5000	16	1	10	10	300	3.05	1.706667	5000	16	1	10	10	300	5.81	23.17333
uf125-051								uf150-051							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	3.07		5000	16	1	10	10	100	1.49	
5000	16	1	10	10	200	2.05		5000	16	1	10	10	200	1.49	
5000	16	1	10	10	300	5.09	3.403333	5000	16	1	10	10	300	1.5	1.493333
uf125-060								uf150-056							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.03		5000	16	1	10	10	100	1.49	
5000	16	1	10	10	200	1.03		5000	16	1	10	10	200	4.39	
5000	16	1	10	10	300	1.03	1.03	5000	16	1	10	10	300	2.94	2.94
uf125-066								uf150-068							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.03		5000	16	1	10	10	100	1.5	
5000	16	1	10	10	200	1.03		5000	16	1	10	10	200	1.5	
5000	16	1	10	10	300	1.03	1.03	5000	16	1	10	10	300	1.49	1.496667
uf125-068								uf150-074							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.03		5000	16	1	10	10	100	1.46	
5000	16	1	10	10	200	1.04		5000	16	1	10	10	200	1.46	
5000	16	1	10	10	300	1.04	1.036667	5000	16	1	10	10	300	1.46	1.46
uf125-075								uf150-081							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.04		5000	16	1	10	10	100	68.5	
5000	16	1	10	10	200	1.04		5000	16	1	10	10	200	5.86	
5000	16	1	10	10	300	1.03	1.036667	5000	16	1	10	10	300	16.06	30.14
uf125-096								uf150-091							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)		Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.04		5000	16	1	10	10	100	18.82	
5000	16	1	10	10	200	1.03		5000	16	1	10	10	200	11.6	
5000	16	1	10	10	300	1.04	1.036667	5000	16	1	10	10	300	5.81	12.07667

Running Time: 125 variables, 538 clauses, 1600 population and 150 variables, 645 clauses, 1600 population

175 variables, 753 clauses, 1600 population							
uf175-019							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	37.91	
5000	16	1	10	10	200	17.96	
5000	16	1	10	10	300	117.63	57.83333
uf175-034							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	17.82	
5000	16	1	10	10	200	140.31	
5000	16	1	10	10	300	13.81	57.31333
uf175-048							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.01	
5000	16	1	10	10	200	9.89	
5000	16	1	10	10	300	9.89	7.263333
uf175-049							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.02	
5000	16	1	10	10	200	2.01	
5000	16	1	10	10	300	7.97	4
uf175-057							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.01	
5000	16	1	10	10	200	2.01	
5000	16	1	10	10	300	5.98	3.333333
uf175-063							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	4.01	
5000	16	1	10	10	200	7.98	
5000	16	1	10	10	300	2.02	4.67
uf175-077							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	19.91	
5000	16	1	10	10	200	4.02	
5000	16	1	10	10	300	35.87	19.93333
uf175-026							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.01	
5000	16	1	10	10	200	3.96	
5000	16	1	10	10	300	2	2.656667
uf175-090							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	1.99	
5000	16	1	10	10	200	1.99	
5000	16	1	10	10	300	1.99	1.99
uf175-099							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.02	
5000	16	1	10	10	200	2.02	
5000	16	1	10	10	300	2.01	2.016667

200 variables, 860 clauses, 1600 population							
uf200-01							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average(s)
5000	16	1	10	10	100	7.81	
5000	16	1	10	10	200	18.21	
5000	16	1	10	10	300	2.62	7.81
uf200-02							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	5.25	
5000	16	1	10	10	200	7.86	
5000	16	1	10	10	300	15.68	7.86
uf200-081							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.64	
5000	16	1	10	10	200	2.64	
5000	16	1	10	10	300	7.87	2.64
uf200-077							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	12.98	
5000	16	1	10	10	200	23.37	
5000	16	1	10	10	300	5.21	12.98
uf200-058							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.63	
5000	16	1	10	10	200	2.63	
5000	16	1	10	10	300	5.23	2.63
uf200-85							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.63	
5000	16	1	10	10	200	2.63	
5000	16	1	10	10	300	2.63	2.63
uf200-094							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	43.15	
5000	16	1	10	10	200	571.93	
5000	16	1	10	10	300	925.41	571.93
uf200-05							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	5.23	
5000	16	1	10	10	200	2.63	
5000	16	1	10	10	300	5.23	5.23
uf200-089							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	319.12	
5000	16	1	10	10	200	172.37	
5000	16	1	10	10	300	375.63	289.04
uf200-069							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	2.62	
5000	16	1	10	10	200	2.61	
5000	16	1	10	10	300	2.61	2.613333

Running Time: 175 variables, 753 clauses, 1600 population and 200 variables, 860 clauses, 1600 population

250 variables, 1065 clauses, 1600 population							
uf250-01							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	20.04	
5000	16	1	10	10	200	83.95	
5000	16	1	10	10	300	23.99	42.66
uf250-044							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	16.06	
5000	16	1	10	10	200	356.68	
5000	16	1	10	10	300	168.32	180.3533
uf250-057							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	x	
5000	16	1	10	10	200	133.57	
5000	16	1	10	10	300	546.53	340.05
uf250-035							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	4.04	
5000	16	1	10	10	200	228.67	
5000	16	1	10	10	300	56.21	96.30667
uf250-068							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	96.86	
5000	16	1	10	10	200	60.56	
5000	16	1	10	10	300	x	78.71
uf250-081							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	x	
5000	16	1	10	10	200	206.33	
5000	16	1	10	10	300	461.13	333.73
uf250-098							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	133.29	
5000	16	1	10	10	200	113.14	
5000	16	1	10	10	300	141.42	129.2833
uf250-074							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	97.34	
5000	16	1	10	10	200	24.4	
5000	16	1	10	10	300	316.27	146.0033
uf250-037							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	111.85	
5000	16	1	10	10	200	20.03	
5000	16	1	10	10	300	28	53.29333
uf250-04							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	
5000	16	1	10	10	100	12.1	
5000	16	1	10	10	200	4.04	
5000	16	1	10	10	300	8.08	8.073333

Running Time: 250 variables, 1065 clauses, 1600 population

uf250-01								uf250-04							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average (s)	Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	Average (s)
5000	8	1	20	10	100	72.35		5000	8	1	20	10	100	39.11	
5000	8	1	20	10	200	73.65		5000	8	1	20	10	200	11.18	
5000	8	1	20	10	300	16.7	54.23333333	5000	8	1	20	10	300	5.61	18.6333333
5000	10	1	16	10	100	78.18		5000	10	1	16	10	100	9.27	
5000	10	1	16	10	200	50.16		5000	10	1	16	10	200	46.26	
5000	10	1	16	10	300	27.64	51.99333333	5000	10	1	16	10	300	41.61	32.38
5000	16	1	10	10	100	20.04		5000	16	1	10	10	100	12.01	
5000	16	1	10	10	200	83.95		5000	16	1	10	10	200	4.04	
5000	16	1	10	10	300	23.99	42.66	5000	16	1	10	10	300	8.07	8.04

Scalability: 250 variables, 1065 clauses, 1600 population

uf250-044							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	average (s)
5000	8	1	20	10	100	1235.58	
5000	8	1	20	10	200	385.94	
5000	8	1	20	10	300	126.92	582.8133333
5000	10	1	16	10	100	91.81	
5000	10	1	16	10	200	211.04	
5000	10	1	16	10	300	431.4	244.75
5000	16	1	10	10	100	16.06	
5000	16	1	10	10	200	356.68	
5000	16	1	10	10	300	168.32	180.3533333

uf250-074							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	average (s)
5000	8	1	20	10	100	198.17	
5000	8	1	20	10	200	11.35	
5000	8	1	20	10	300	107.42	105.646667
5000	10	1	16	10	100	168.21	
5000	10	1	16	10	200	84.19	
5000	10	1	16	10	300	425.09	225.83
5000	16	1	10	10	100	97.34	
5000	16	1	10	10	200	24.4	
5000	16	1	10	10	300	316.27	146.003333

Scalability: 250 variables, 1065 clauses, 1600 population

uf250-098							
Maximum Tries	Grid.x	Grid.y	block.x	block.y	seed	Second (s)	average (s)
5000	8	1	20	10	100	627.07	
5000	8	1	20	10	200	39.82	
5000	8	1	20	10	300	188.06	284.9833333
5000	10	1	16	10	100	65.45	
5000	10	1	16	10	200	x	
5000	10	1	16	10	300	206.4	135.925
5000	16	1	10	10	100	133.29	
5000	16	1	10	10	200	113.14	
5000	16	1	10	10	300	141.42	129.2833333

Scalability: 250 variables, 1065 clauses, 1600 population