國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

運用計算統一設備架構實現之平行化布爾可滿足性解法器

Implementation of Parallel Boolean Satisfiability Solver by CUDA (Compute Unified Device Architecture)

林拱民

Kung-Ming Lin

指導教授：黃鐘揚 博士

Advisor: Chung-Yang (Ric) Huang, Ph.D.

中華民國 100 年 6 月

June, 2011

誌謝

感謝指導教授黃鐘揚老師兩年來對我的耐心與包容、感謝吳濟安學長於布爾可滿足性解法器領域提供的知識與討論。

感謝已在天國的父親的深遠影響、感謝母親多年以來無私的付出、感謝舍妹沒有形諸言語的支持。

感謝每一位陪我度過兩年碩士班生涯的人。

感謝你們，讓我的生命歷程變得更加美好。

林拱民　中華民國一百年六月　於臺灣大學電機二館 353 設計驗證實驗室

中文摘要

解決布爾可滿足性問題在理論與工業應用上，扮演著極為關鍵角色。十五年來，布爾可滿足性解法器有著長足的進展，得以解決相當大型的問題。為了進一步解決更大型、更困難的布爾可滿足性問題，平行化布爾可滿足性解法器於近年來受到相當關注。最近幾年的國際布爾可滿足性解法器比賽之中，最佳的四至八線程平行化布爾可滿足性解法器的成績勝過最佳單線程布爾可滿足性解法器。

通用計算於圖形處理單元也在大量平行運算的領域之中興起。為了探討大量平行化布爾可滿足性解法器的概念，我們運用計算統一設備架構的平台，實現了名為「CUDASAT」的子句分享平行化、衝突驅使子句學習、戴維斯-普特南-羅格曼-羅夫蘭演算法布爾可滿足性解法器。

根據我們瞭解，目前尚未有類似 CUDASAT 的程式。實驗結果顯示，提昇平行解法器的數量時，平均每個解法器於搜尋上所用的步驟有著明顯下降的趨勢。CUDASAT 的效能無法與現今最好的平行化布爾可滿足性解法器相比。它是作為發展大量平行化、低成本、替代性布爾可滿足性解法器的原型。

關鍵字：布爾可滿足性問題、可滿足性解法器、平行運算、衝突驅使子句學習、戴維斯-普特南-羅格曼-羅夫蘭、通用計算於圖形處理單元、計算統一設備架構。

Abstract

Boolean satisfiability (SAT) problem plays a critical role in theoretical and industrial applications. With the advance of SAT solvers in the past 15 years, we are capable to solve fairly large-scale problems. To improve the performance of SAT solvers for much larger and harder SAT problems, parallelization of SAT solvers is gaining much attention in recent years. The state-of-the-art 4-to-8 threaded parallel SAT solvers are more powerful than single-threaded ones in recent international SAT solver competitions.

General-Purpose computation on Graphics Processing Units (GPGPU) is also emerging from massive parallel computing realm. To explore the concept of massive parallel SAT solvers, we have implemented the "CUDASAT", a parallel CDCL-DPLL (*Conflict Driven Clause Learning - Davis-Putnam-Logemann-Loveland*) SAT solver with clause sharing on CUDA (*Compute Unified Device Architecture*) platform.

To the best of our knowledge, CUDASAT is the first of its kind. The experimental results demonstrated a downward trend in average searching events per solver while increasing the number of parallel solver. While the performance is not comparable to those state-of-the-art parallel SAT solvers, CUDASAT serves as a prototype of massive parallelization toward an affordable and alternative solution for SAT solving.

Keywords: Boolean satisfiability problem, satisfiability solver, parallel computing, SAT, CDCL, DPLL, GPGPU, CUDA.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Boolean satisfiability (SAT) problem plays a critical role in theoretical and industrial applications. The advancement of SAT solvers over the past 15 years is significant. Most modern state-of-the-art SAT solvers are based on DPLL algorithm [1] [2]. In addition of DPLL framework, the performance and scalability of modern SAT solvers are greatly enhanced by the following four techniques:

(1) *Conflict-driven Clause Learning* (CDCL) [3] [4] [5].

(2) Restart of search process [6].

(3) Lazy data structures e.g. two-watched-literal scheme for Boolean constraint propagation [7].

(4) Adaptive branching based on conflicts, i.e. non-chronological back-tracking [7].

The performance of SAT solvers is still improving, but without a great acceleration in past few years. With the dominance of multi-core CPUs, traditional single-threaded SAT solvers start to shift their strategies into the realm of parallelization. Modern parallel SAT solver launches multiple instances of single-threaded solvers with inter-solver cooperation and management. The first award winner *plingeling* [8] in SAT-Race 2010 Special Track 1 (CNF parallel) beats the first award winner *CryptoMiniSat* [9] in Main Track (CNF sequential) by the number of solved cases and the average time per solved instance. Eight CPU cores are available in SAT-Race 2010 Special Track 1 (CNF parallel).

For SAT solvers, the jewel in the crown is speed while they must report correct satisfiable/unsatisfiable results. The faster the solver, the more likelihood it could find a solution or conclude there is no solution within a reasonable time. All single-threaded SAT solvers are pursuing a higher speed by more advanced, more complex, more complete, and faster pre-processor and in-processor. Pre-processing simplifies the problem before solving it by self-subsuming resolution, variable elimination, subsumption, etc. In-processing further simplifies the problem during the process of solving by using learnt clauses, doing asymmetric branching, binary clause reasoning, variable replacement, blocked clause elimination, failed literal probing, tautology elimination, clause strengthening with tautological binary clauses, clause cleaning, etc. [10] according to Mate Soos, the author of *CryptoMiniSat*.

The number of solvers in a parallel SAT solver is limited to how many CPU cores are available on a server with multi-core CPUs. The most common server configuration is two quad-core CPUs, which has 8 cores in total. What could we expect as the number grows up? Supercomputers or computer grids are both equipped many CPUs by different architectures. They are ideal for running a large amount of SAT solvers concurrently. While we have no access to such expensive and scarce computing resource, we take an affordable graphic processing unit (GPU) as our experimental platform. A GPU is capable of launching massive threads natively, with a much lower computing speed and very limited memory capacity compared to a CPU. To explore the scalability and effect of SAT solver parallelization, we implemented CUDASAT, a CDCL-DPLL parallel SAT solver with clause sharing on CUDA [11] from scratch.

We observed a downward trend in average conflicts, decisions, propagations, and

conflict literals per solver while increasing SAT solver instances from one to 256. The statistics of events indicates how much effort a SAT solver is spent before finding a solution or concluding there is no solution. A lower event number in searching process represents a lesser effort used in a solver. If we could run all those solvers on CPU cores exclusively, more solvers would result in less runtime while they are taking less effort in average per solver.

To the best of our knowledge, CUDASAT is the first parallel DPLL-CDCL SAT solver implemented on CUDA. CUDASAT is an exploratory prototype on GPU for massive parallelization of SAT solvers. As the advancement on GPGPU is gaining a great momentum in recent years, CUDA might become an affordable alternative solution for massive parallel SAT solving in the future.

Chapter 2 Previous Work

2.1 The Categories of SAT Solvers

Satisfiability solvers could be categorized by their heuristics:

(1) *Complete*: complete methods guarantee to find a satisfying solution or to report that no solution exists (unsatisfiable). These methods include existential quantification, inference rules, systematic search, and a combination of search and inference. Almost all modern complete solvers are based on a systematic search DPLL algorithm [1] [2]. DPLL is a complete *depth first search* (DFS) algorithm in the problem's solution space. There are other commercial solvers based on patented [12] Stålmarck's method [13] [14]. It is a natural deduction proof system more like a *breadth first search* (BFS) algorithm. They could deal with satisfiable and unsatisfiable problems.

(2) *Stochastic* (*incomplete*): incomplete methods do not guarantee to find a satisfying solution or report that no solution exists. Incomplete solvers are generally based on *stochastic local search* (SLS), survey propagation, simulated annealing, etc. There are many researches about improving this method. For example, GSAT [15] and WalkSAT [16] use a greedy local search algorithm in a given problem's solution space. They generally work better in random generated SAT problems. State-of-the-art incomplete solvers could be found in SAT Competition 2009 [17] category random instances (RANDOM)

such as TNM, gnovelty+2, adaptg2wsat2009++. The detailed description of these solvers could be found in [9].

(3) *Hybrid*: some solvers incorporate the complete and incomplete methods. Several hybrid solvers like SATzilla2009_R and hybridGM3 could be found in [9]. Like (1), they are complete solvers.

(4) *Parallel*: state-of-the-art parallel SAT solvers launch multiple solver instances concurrently. The winners of SAT-Race 2010 [18] Special Track 1: Parallel (multi-threaded) SAT solvers are plingeling [8] and ManySAT [19]. Most of them are multi-thread DPLL-CDCL solvers. Some parallel solvers are based on combining DPLL and Stålmarck's method [20] or SLS [21].

There are also some researches on hardware SAT solvers on field programmable gate array (FPGA) chips like [22], which are based on DPLL-CDCL algorithm.

2.2 Parallel SAT solvers

Current state-of-the-art SAT solver heuristics fall into two categories. One is DPLL (*Davis-Putnam-Logemann-Loveland*) [1] [2] -CDCL (*Conflict-Driven Clause Learning*) [5] SAT solvers. CDCL SAT solvers commonly involves many heuristics, the detailed list could be found in [5], p132. They are quite successful in structural SAT problems, which is usually a conversion of real world problems into the form of Boolean satisfiability problem. The other is stochastic SAT solvers. They are good at random

generated problems but weak in other realms.

DPLL algorithm is sequential by its nature. Each decision of variable assignment must be taken one after another. These decisions form a path in the decision tree of a satisfiability problem. One DPLL SAT solver is always sequential.

Stochastic SAT solver algorithms heavily rely on high speed pseudo-random number generation. In principle, all procedures could be parallelized naturally.

Current parallel DPLL-CDCL solvers have several solver instances running concurrently. Each solver is essentially sequential. Thus “parallel” here means running multiple SAT solvers at the same time, not a parallel algorithm in SAT problem solving.

The state-of-the-art DPLL-CDCL SAT solvers utilize many heuristics to find “hints” in a given problem. These hints may guide them to several “short-cuts” to a solution or unsatisfiable core in the structure of a given problem. An essential heuristic is VSIDS (*Variable State Independent Decaying Sum*) introduced by Chaff [7] SAT solver. It is a dynamic variable ordering based on past activities. The “activity” of variables and clauses are increased when they are involved in a conflict. High activity may indicate they are important in this problem by past decisions and conflicts. VSIDS chooses the highest activity variable as the next decision variable to assign, and removes low activity learnt clauses for performance gain.

DPLL-CDCL SAT solvers lose their performance in random generated problems because there are no reasonable or traceable “hints” in these problems. Thus stochastic

SAT solvers often out-perform them in this realm by totally different strategies. But it is not enough to use solely SLS algorithms to solve random SAT problems efficiently. The winners in recent international SAT solver competitions random track are all hybrid SAT solvers.

2.2.1 Plingeling and ManySAT

Plingeling [8] and ManySAT [19] are the first and the second rank parallel SAT solvers in SAT-Race 2010 [18]. They are both portfolio-based parallel SAT solver. Portfolio means a combination of different SAT solvers in their strategies and parameters.

ManySAT is the first-rank of parallel solvers in SAT-Race 2008. It employs 4 instances with different carefully human-crafted heuristics and parameter-tunings. To increase the diversity of decisions, each solver has 2% ~ 3% random noises in VSIDS heuristic. All solvers exchange learnt clauses with their size less or equal to 8.

Plingeling is a multi-threaded version of Lingeling [8]. Solver instances have different pre-processing algorithms and default orderings for variable ordering and indices used for initialization and tie-breaking in decision heuristics by different random seeds. Unlike ManySAT, all instances in plingeling only exchange unit clauses.

2.2.2 Decision Divergence and Learnt Clauses Exchange

The CPU cores available on a high-end server are usually 8. They are often two quad-core CPUs linked by a high-speed bus on the mainboard. The power of parallelization of sequential DPLL-CDCL solvers comes from solver decision divergence or learnt clauses exchange.

With decision divergence, each solver could explore different decision path and learn different information about the local structure of problem. This information is represented in the form of learnt clauses. Exchanging learnt clauses enables all solvers to know unseen information gathered by other solvers. Thus they won't make the same mistake (i.e. conflict) in the future.

2.3 SAT Solvers on CUDA

Some CUDA SAT solvers found in literatures are listed below:

1. In [23], MESP (MiniSAT enhanced with Survey Propagation) is a *survey propagation* [24] [25] based method to modify the variable ordering heuristic in MiniSAT [26]. Survey propagation is an iterative "message-passing" algorithm designed to solve high-density random k-SAT problems. Survey propagation is implemented on CUDA as a guide to modify MiniSAT's variable ordering heuristic *variable state independent decaying sum* (VSIDS) heuristics. They have reported a 2.35x to 2.42x speed-up in average.

2. In [27], a 3-SAT solver using divide and conquer recursive searching is implemented purely on CUDA. They have demonstrated a SAT solving technique targeted on random SAT problems utilizing massive parallelism. This solver runs on a problem scale of 75 variables and 325 clauses.

3. In [28], GPU4SAT is an incomplete SLS SAT solver implemented on CUDA 1.1. This solver runs on a scale of 256 variables and 1088 clauses.

Other works could be found by searching keywords “CUDA” and “SAT”. There are some unfinished or empty projects on Google code [29] like “abrasatcuda” and “cuda-sat”. Mate Soos, the author of CryptoMiniSat, had created a CUDA branch in his developmental code. He had done a preliminary work and stopped in September 2010.

To the best of our knowledge, no parallel DPLL-CDCL SAT solver has been successfully implemented on CUDA. In each SAT solver, the most part of DPLL-CDCL algorithm could not be parallelized. Due to the inferior computing power of each CUDA core, it seems reasonable to predict that no speed-up could be gained from CUDA DPLL-CDCL SAT solvers. The goal of our CUDASAT is to explore the massive parallelism of parallel SAT solvers.

Chapter 3 Preliminaries

CUDASAT is a parallel Boolean satisfiability solver implemented on CUDA. We introduce some basic knowledge about Boolean satisfiability and CUDA before getting into the details of CUDASAT.

3.1 Boolean Satisfiability

3.1.1 Definitions

We only state mathematical definitions related to Boolean satisfiability problems.

- **Boolean algebra:**

  A *Boolean algebra* is an algebraic structure:

$$(B, \vee, \wedge, \neg, 0, 1)$$

B is a set. $\vee$ and $\wedge$ are binary operations defined on *B*, which are disjunction and conjunction respectively. 0 and 1 are distinct members of *B*. A variable *x* in Boolean algebra could be only two values: 0 (*false*) or 1 (*true*).

■ **Basic operations:**

(1) Conjunction (AND): is a binary operation. Conjunction is often written as $x \wedge y$, $x \cdot y$, or xy. The values of $x \wedge y$ are listed below:

	$\wedge$	y	
		0	**1**
x	**0**	0	0
	1	0	1

Table 3.1 - Truth table of conjunction operation.

(2) Disjunction (OR): is a binary operation. Disjunction is often written as $x \vee y$ or $x + y$. The values of $x \vee y$ are listed below:

	$\vee$	y	
		0	**1**
x	**0**	0	1
	1	1	1

Table 3.2 - Truth table of disjunction operation.

(3) Complement or negation (NOT): is a unary operation. Complement is often written as $\neg x$, $\bar{x}$, or x'.The values of $\neg x$ are listed below:

	$\neg$	
x	**0**	1
	1	0

Table 3.3 - Truth table of complement operation.

■ **Axioms of Boolean algebra:**

(1) B is *closed* under $\vee$, $\wedge$, and $\neg$.

$\forall x, y \in B$,

$x \vee y \in B$ and $x \wedge y \in B$.

$\forall x \in B$,

$\neg x \in B$.

(2) Commutative laws:

$\forall x, y \in B$,

$x \vee y = y \vee x$ and $x \wedge y = y \wedge x$.

(3) Distributive laws:

$\forall x, y \in B$,

$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.

(4) Identities:

$\forall x \in B$,

$0 \vee x = x$ and $1 \wedge x = x$.

(5) Complements:

$\forall x \in B$, $\exists \neg x \in B$ such that

$x \vee \neg x = 1$ and $x \wedge \neg x = 0$.

■ **De Morgan's Laws:**

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\neg(x \vee y) = \neg x \wedge \neg y$$

- **Conjunctive normal form (CNF):**

A CNF formula φ with n Boolean *variables* $v_1, \ldots, v_n$ is the conjunction (AND) of m *clauses* $c_1, \ldots, c_m$. Each clause is the disjunction (OR) of at least one *literal*. A literal is a variable or its complement. A Boolean formula φ is a unique Boolean function f with n variables. Each clause is an implicate of f.

- **Boolean satisfiability problem:**

The Boolean satisfiability (SAT) problem is concerned with finding a set of variable assignments to the arguments of $f(v_1,...,v_n)$ that makes f equal to 1 (true) or proving f is equal to the constant 0 (unsatisfiable).

An example is given below:

$$\varphi_1 = f(v_1,...,v_4) = (v_1 + \overline{v_3})(\overline{v_1} + v_2 + v_4)(v_3 + v_4)(\overline{v_2})$$

v_i is a literal with positive *polarity* and $\overline{v_i}$ is a literal with negative polarity. $(v_1 + \overline{v_3})$ is a clause. A satisfiability problem is whether we could find a set of variable assignments to make φ_1 equals to 1 or not. φ_1 is satisfiable for $(v_1, v_2, v_3, v_4) = (1, 0, 0, 1)$. Please note that there are other satisfiable solutions for φ_1.

For another example φ_2 below:

$$\varphi_2 = f(v_1,...,v_4) = \left(v_1 + \overline{v_3}\right)\left(\overline{v_1} + v_2 + v_4\right)\left(v_3 + v_4\right)\left(\overline{v_2}\right)\left(\overline{v_1} + v_3 + \overline{v_4}\right)\left(v_1 + v_3 + \overline{v_4}\right)\left(\overline{v_1} + \overline{v_3} + \overline{v_4}\right)$$

No assignment could be found for φ_2 to be 1, thus φ_2 is unsatisfiable.

Cook–Levin theorem [30] [31] states that the Boolean satisfiability problem is NP-complete. It is the first NP-complete problem proven by Cook in 1971.

3.1.2 SAT Applications

The satisfiability problems are critical in logic, graph theory, computer science, engineering, operation research, and other areas. Many problems could be transformed into satisfiability problems. Solving SAT problems poses both theoretical and practical interests. Some examples of satisfiability applications are: theorem proving, statistical physics, bounded model checking, AI planning, software and hardware verification and modeling, combinational circuit designs, automatic test pattern generation (ATPG), circuits equivalence checking, property checking.

Though satisfiability problems are NP-complete based on the worst-case complexity analysis, not all satisfiability problems are hard in average. In practice, modern SAT solvers could solve many satisfiability problems efficiently. Efficient satisfiability solving stands at the center of many real world engineering problems.

3.2 Compute Unified Device Architecture (CUDA)

Graphic Processing Unit (GPU) is a computing device dedicated to rendering graphics. It is a highly parallel architecture for computing each pixels and massive stream of data for graphics and video.

General-Purpose computation on Graphics Processing Units (GPGPU) is a concept of exploit computation on GPU's many-core processors other than rendering graphics. It is not possible before the graphics card manufactures adding programmability and high precision arithmetic to their hardware architecture. Hacking GPUs for computing is as early as the early 2000s, much earlier before the official software development kits (SDK) are released. AMD/ATI announced Stream in 2006, while NVIDIA's CUDA was in later 2006. We choose CUDA for its friendlier application programming interface (API) and SDK.

3.2.1 Hardware Architecture

GPU is specialized in parallel processing massive data streams. It has a many-core design architecture, with each core is capable executing light-weighted instructions. On GTX 480 card, it has 480 cores for integer and floating point arithmetic. It is capable of running 23040 threads concurrently.

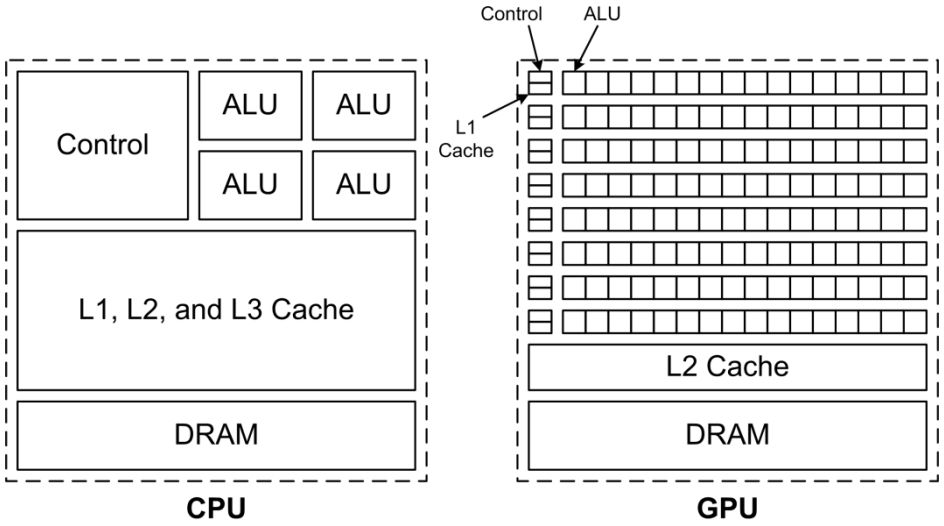


Figure 3.1 - Multi-core CPU vs. many-core GPU hardware (modified from [11])

3.2.2 Programming Model

CUDA is an extension of C/C++ programming language, while the portion of C/C++ is not fully supported. It has three key abstractions for parallelization:

(1) Hierarchy of threads: threads are grouped into blocks, and blocks are grouped into grids.

(2) Shared memory: a user controlled cache memory, which is small but much faster than the large on-card DRAM.

(3) Barrier synchronization: various degrees of synchronization could be issued as needed.

- **Kernels**

Kernels are user defined C function like code, CUDA executes them by *N* copies,

N CUDA threads in parallel. The following code adds two vectors *A* and *B* of size *N* and stored the result into vector *C*: (modified from [11])

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

int main ( void )
{
    …
    // Kernel invocation with N threads, single block.
    vec_add <<< 1, N >>> ( A, B, C );
}
```

- **Thread Hierarchy**

All threads and blocks have their own 3 dimensional indices. All threads in a block share the limited size shared memory on one hardware core. The shared memory is a low-latency memory near each processor core (48KB on Fermi architecture cards). All blocks are independently executed in any order. Though inter-block synchronization commands are available in CUDA, they do not synchronize blocks exactly as a user expected. In practice, inter-block synchronization is only achievable by turning off L1 cache.

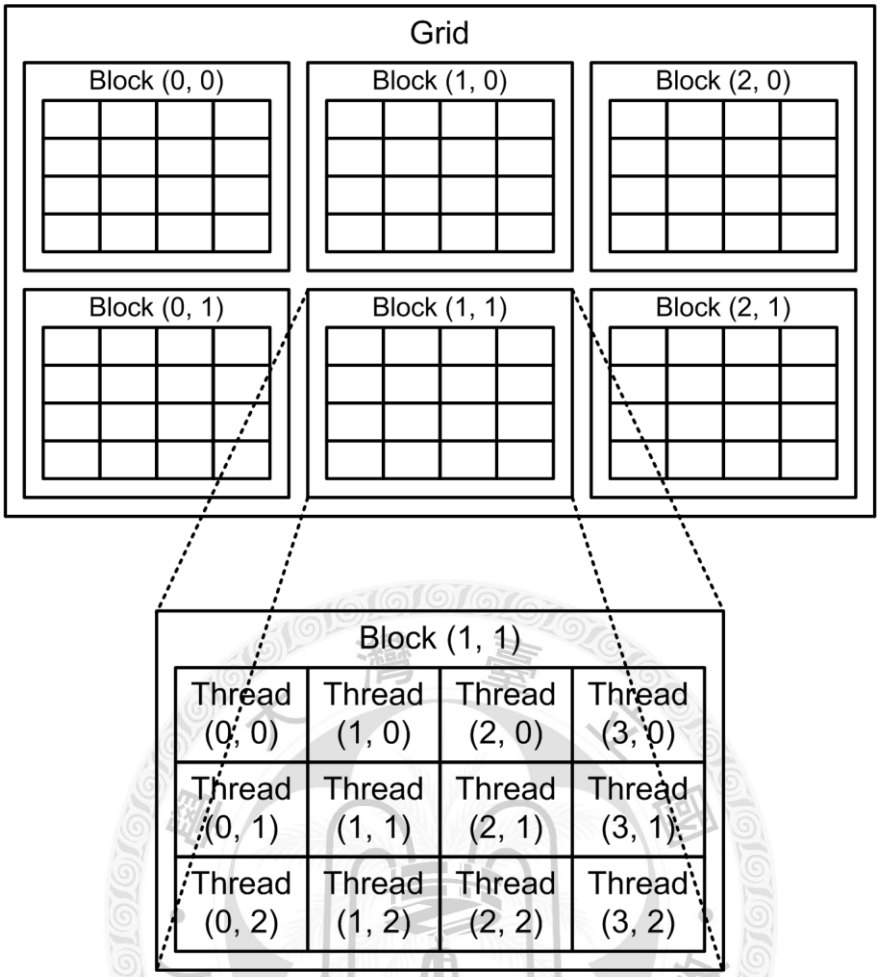


Figure 3.2 - The thread hierarchy seen by a user (modified from [11])

- **Warps**

CUDA groups 32 parallel threads into warps. Each thread in a warp starts at the same program address, but they have their own instruction address counters and register states. They are free to branch and execute independently.

All 32 threads in a warp execute one common instruction at a time. If threads in a warp diverge via a data-dependent conditional branch, the warp executes each branch path in serial, disabling threads that are not on that path. When all paths complete, the

threads converge back. Branch divergence occurs only within a warp.

In practice, high warp divergence causes low performance due to serialization of divergent executional path.

- **Memory hierarchy**

Each thread has its private local memory. Local variables reside in registers on chip. If the number of local variables exceeds the number of registers available, they are spilled into global memory. Each thread block has shared memory (user controlled cache) visible to all threads in the block and with the same lifetime as the block. All threads have access to the global memory.

The constant memory is a read-only cache for threads on device. The texture memory is a read-only space on global memory but with special hardware management optimized for different access mode.

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

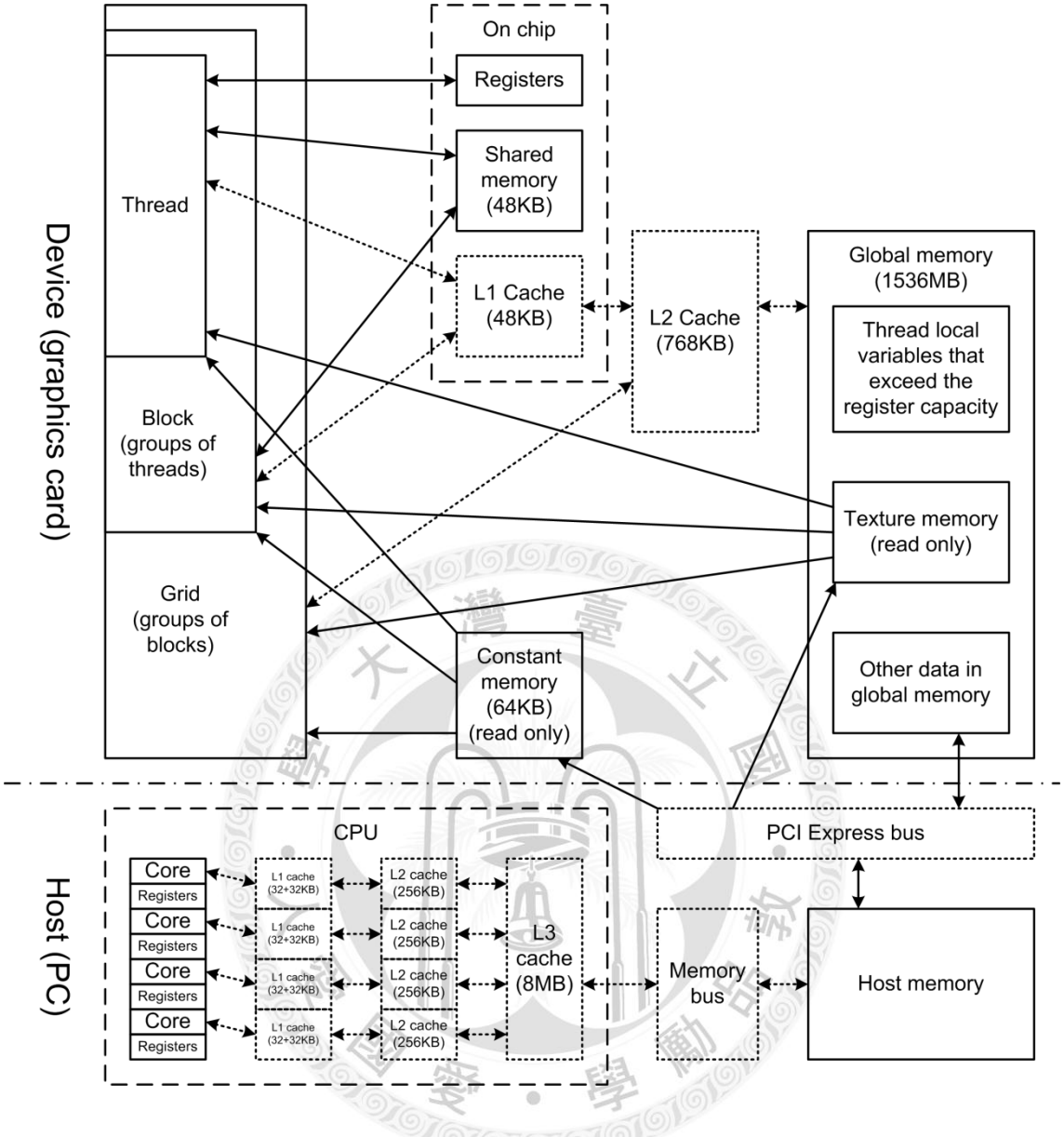


Figure 3.3 - CPU (Intel i7) and CUDA (NVIDIA Fermi) memory hierarchy

- **Processing flow:**

CUDA assumes that the threads execute on a physically separate device that operates as a coprocessor to the host running the C program. It also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the

global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and de-allocation as well as data transfer between host and device memory.

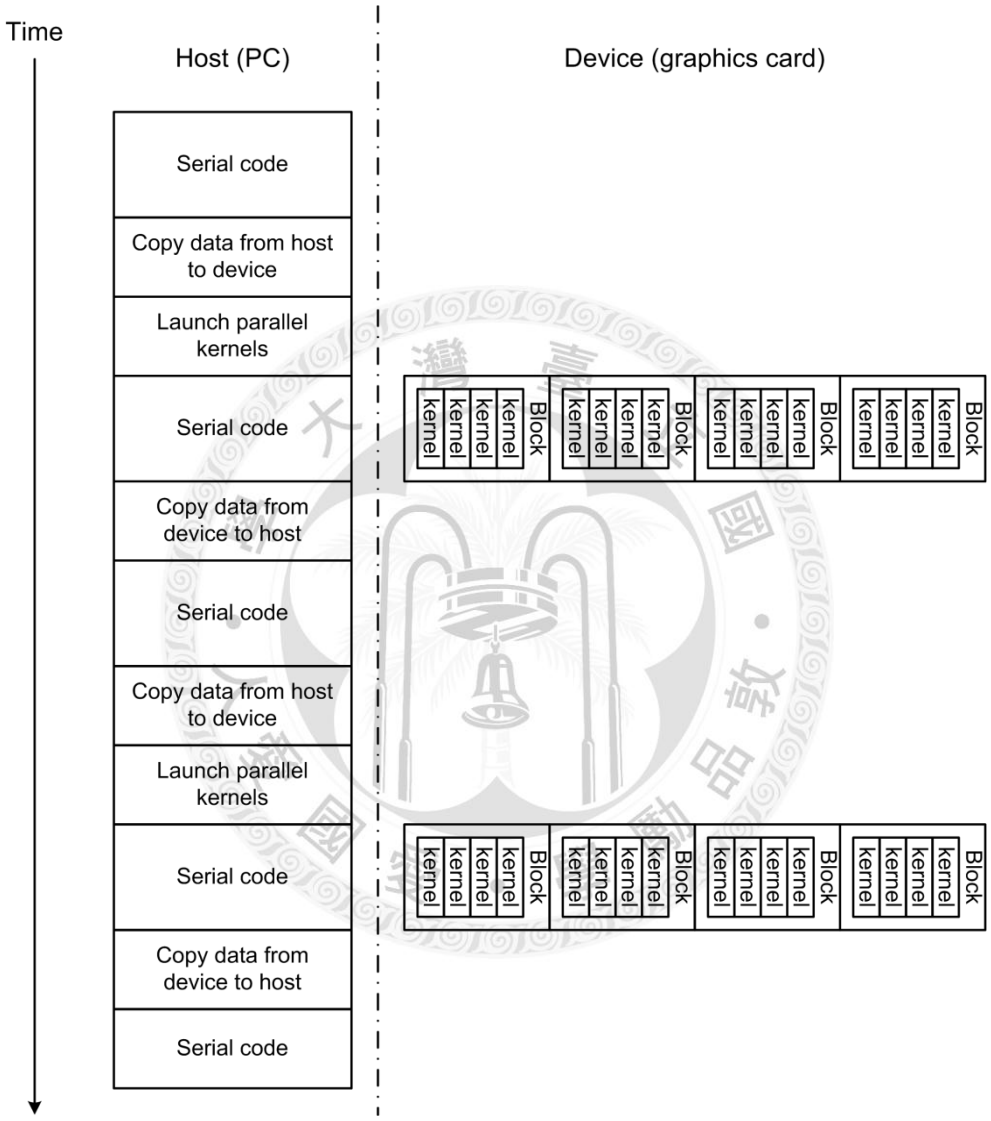


Figure 3.4 - Typical CUDA processing flow

- **Typical CUDA practice**

To get the best parallel computing power of CUDA, a programmer must take care of a lot of hardware and optimization details. Here is a good CUDA practice in general:

(1) Copy all input data from host to device global memory. The data transfer between host and device is through PCI Express bus, thus the number of copy commands should be as less as possible. Use texture memory to exploit the data locality when possible. Try not to use dynamic device memory allocation, they are extremely slow.

(2) Copy the input data from global memory to the user controlled shared memory in a coalescence style. Since shared memory is small but much faster, it is best to process a part of input data in shared memory. The memory access pattern is extremely important for CUDA performance. Access coalescence should be on the top priority. Because of every global memory access request has a very high latency about 400 ns.

(3) Compute the partial input data. Execution flow divergence within a warp is another critical issue for high performance. Beware of possible bank conflicts related with access pattern in shared memory. In Fermi architecture, L1 cache synchronization could be a monstrous problem.

(4) Copy the result to global memory. Fetch the next part of input data.

(5) When all computations are done, copy the output data from device global memory to host.

Unfortunately, CUDASAT suffers from a scattered memory access pattern and a very divergent control flow in nature due to DPLL-CDCL algorithm, along with extremely slow dynamic memory allocation/de-allocation which is necessary for learnt clauses.

Chapter 4 CUDASAT

4.1 Overview of CUDASAT

4.1.1 Massive Parallelization

The best parallel SAT solvers have at most 8 or 12 instances of sequential DPLL-CDCL SAT solvers for how many CPU cores available. Each solver has to fully occupy one CPU core. One CPU core is designed for only one thread to get the maximum computational performance. Multiple threads at one CPU core introduce a huge overhead for managing and switching between different threads when they all demand large amount of computing resource. While one CPU core has to be reserved for only one solver, the scale of parallelization is limited by the amount of CPU cores.

What if “massive” parallelization becomes possible? Not 8 or 16 threads. “Massive” means a parallelization of running 128, 256, or more solvers at the same time. The question we want to answer is:

“Do we get more benefit from more solvers?”

4.1.2 GPGPU vs. CPU

To explore the scalability of a parallel SAT solver, we face two hardware choices. One is CPU, the other is GPGPU (General-Purpose computation on Graphics

Processing Unit). We do not consider supercomputers, computer clusters, and grid computing for that we have no access to such resources. A previous work used grid computing is in [32]; another work used clusters is in [33].

For CPU, multi-tasking is a common technique in operating systems for a long history. Multi-threaded C/C++ programming could be easily implemented by pThread [34], OpenMP [35], or other multi-threading libraries. If we run 256 threads on a 3 GHz quad-core CPU, each thread gets only 47 MHz clock cycles. When we consider the overhead from managing and switching these threads, each solver might get less than 45 MHz of computing power. In modern CPUs, sophisticated flow control and predictions, pipelines, large caches and low memory latency also contribute a great performance gain. Heavy multi-threading causes frequent context switching in pipelines and caches, result in catastrophic performance degradation.

GPGPUs are dedicated for massive parallelism. Their architecture is designed for large amount of light-weighted thread execution and full utilization of parallelism. The mainstream GPGPU development platforms is CUDA (developed by NVIDIA), Stream (developed by AMD/ATI), and OpenCL (Open Computing Language, developed by Khronos Group.) We choose CUDA [11] for its C/C++ like language, friendly API, and supportive SDK.

Though CUDA could execute a large amount of threads (up to 23040 threads concurrently in GTX 480), the performance bottleneck lies within the slow instruction speed and extremely high memory latency. A Fermi architecture [36] card executes one instruction per 4 cycles. For a GTX 480 card with clock speed of 1400 MHz, Each

instruction runs at the speed of 350 MHz. It is at least 17 times slower than a 3 GHz CPU from the view of clock speed. As we take the different hardware design between CPU and GPU into consideration, the computational power of one thread in GTX 480 is even weaker. The memory latency of GTX 480 GDDR5-VRAM is about 400 ns per access, much slower than the 12 ns latency of CPU-to-DDR3-DRAM access.

Both GPGPU and CPU have their own performance downside in massive parallelization of SAT solvers. It is very difficult to assess which one is better. Both of them lead to an extremely low performance for massive parallel SAT solver. As a pioneering exploration, we choose CUDA for its natural parallelization hardware architecture.

4.1.3 CUDASAT: A Parallel SAT Solver on CUDA

The best practice of CUDA is to fully use the parallelism of threads and coalescence of memory access to hide its huge memory latency. The most important goal of parallelism is the independency of computation. The huge memory latency is virtually reduced by fewer accesses to a large bulk data from a continuous memory block.

Unfortunately, the nature of DPLL algorithm forbids the best practice of CUDA. In each instance of solver, all computations are strictly sequential. While all data accesses are scattered, it results in a very large overhead.

Though with these difficulties, we decide to implement a simple DPLL-CDCL SAT

solver on CUDA. To lower the need of computational power, each solver is “simple” compared with those “complex” ones in state-of-the-art SAT solvers. A horde of simple solvers may realize the principle of “collective intelligence.” [37]

CUDASAT is a collection of simple DPLL-CDCL SAT solvers with different random static variable ordering to ensure their decision divergence. With learnt clause sharing, all local conflicts from different local solution space encountered could be combined into a global sense. We describe the details of CUDASAT in the following sections.

4.2 Program Overview

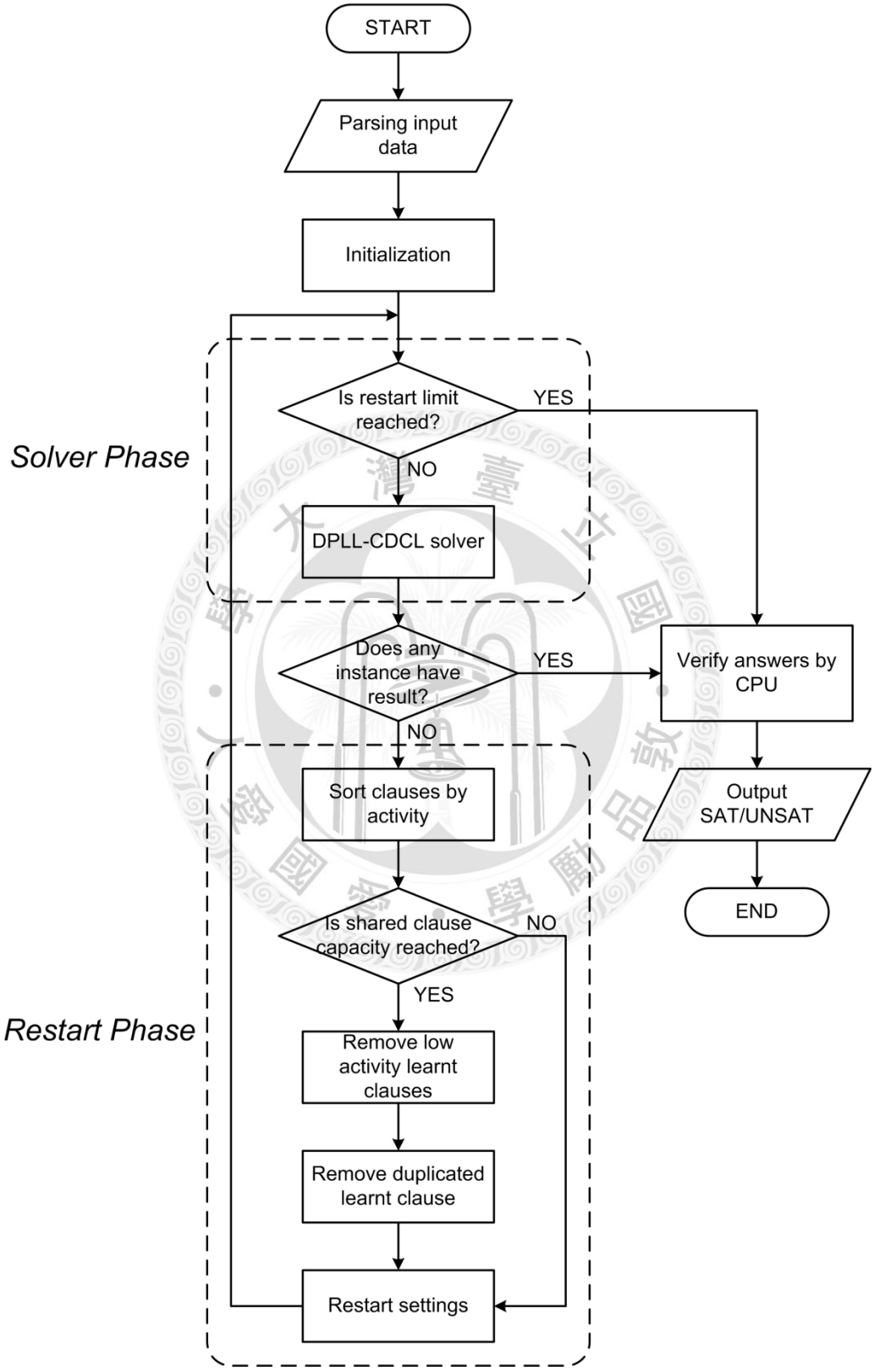


Figure 4.1 - Program flow of CUDASAT.

CUDASAT consists of two phases: solver and restart phases.

In solver phase, every DPLL-CDCL SAT solver tries to find a satisfiable assignments from the first variable to current variable by pre-determined variable ordering. Each variable assignment is decided one after another. If all variables are assigned without conflicts, a solution is found. If no solution could be found, the problem is unsatisfiable. For each decision, it may cause a conflict together with previous decisions. The reason of wrong decision may come from any decision done before. Thus it analyzes the origin of conflict then back-track to the lowest level of decision without conflicts. It continues to do next decision without conflicts. If the conflict limit is reached, it terminates solver phase and enters restart phase.

In restart phase, it checks all learnt clauses for duplicated ones. After removing those duplicated clauses, it prepares for the next solver phase.

The code written like a C/C++ function running on graphics cards is called a kernel. It represents a block model in CUDA. There could be multiple kernels running concurrently on one or more cards. The collection of blocks is called a grid in CUDA. If the number of blocks exceeds the capacity of hardware, CUDA uses a very complicated execution model to make them looks like they are running concurrently. They are not simply waiting in a line to be executed, thus the execution behavior is not deterministic.

The parallel part of CUDASAT is data initialization, removing duplicated learnt clauses, sorting learnt clauses by activity, and data settings before restart all solvers. For solvers, we launch multiple copies of kernels with different variable ordering as parallel

SAT solvers. In CUDASAT, all solver kernels must be halted before processing shared learnt clauses.

There are three configurations to halt all solvers for restarting: (1) all solvers must reach their conflict limit; (2) a solver reaches its conflict limit; (3) a half of solvers reach their conflict limit.

(1) When N solvers reach their conflict limit L for restarting, they gather $N \times L$ leant clauses. They obtain the most complete information during solver phase. But before the last solver reaching its limit, other solvers are leaving the GPUs idling. With the most complete information, the number of restart phases may be reduced at the cost of increased time of solver phases.

(2) As one solver reaches its conflict limit, all solvers are halted immediately for restarting. This configuration prevents GPUs from idling, but gathers the least learnt clauses. With the least information, the number of restart phases may be increased while the time of solver phases decreased.

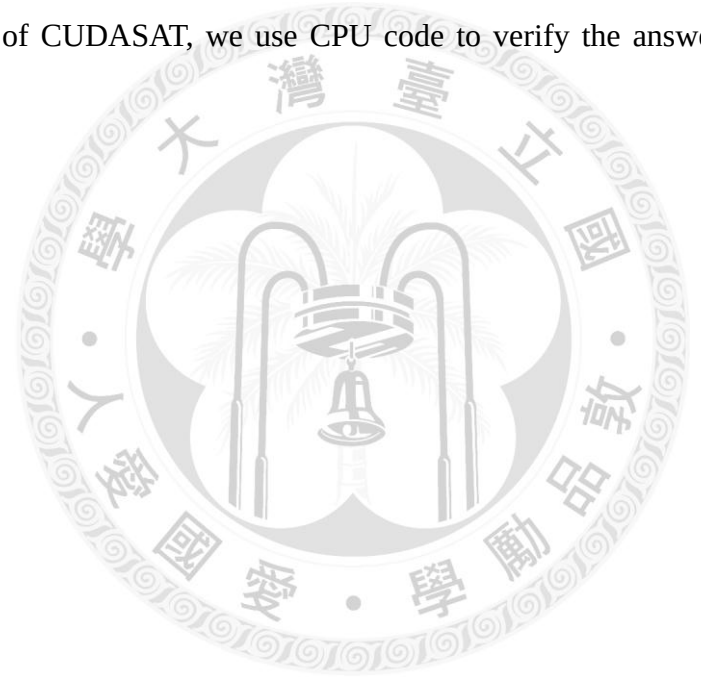
(3) If all solvers halt for restarting when $\frac{N}{2}$ solvers reach their conflict limit L, only $\frac{N}{2}-1$ solvers are idling. The amount of learnt clauses is intermediate. The amount of time spent in solver and restart phases is between configurations (1) and (3). The amount of learnt clauses may also be intermediate.

In CUDA, the minimal parallel unit is a warp, which is composed of 32 threads.

Every thread in a warp executes virtually the same instruction with different data. In CUDASAT, each sequential SAT solver has 32 threads, but most part of SAT solver code is sequential. We only use parallelization in data initializing, settings, BCP (Boolean constraint propagation), and back-tracking.

Due to some synchronization issues, we still use 32 threads in restart phases. This part of code could take more degree of parallelization in the future.

At the end of CUDASAT, we use CPU code to verify the answer given by SAT solver kernels.



4.3 Solver Phase

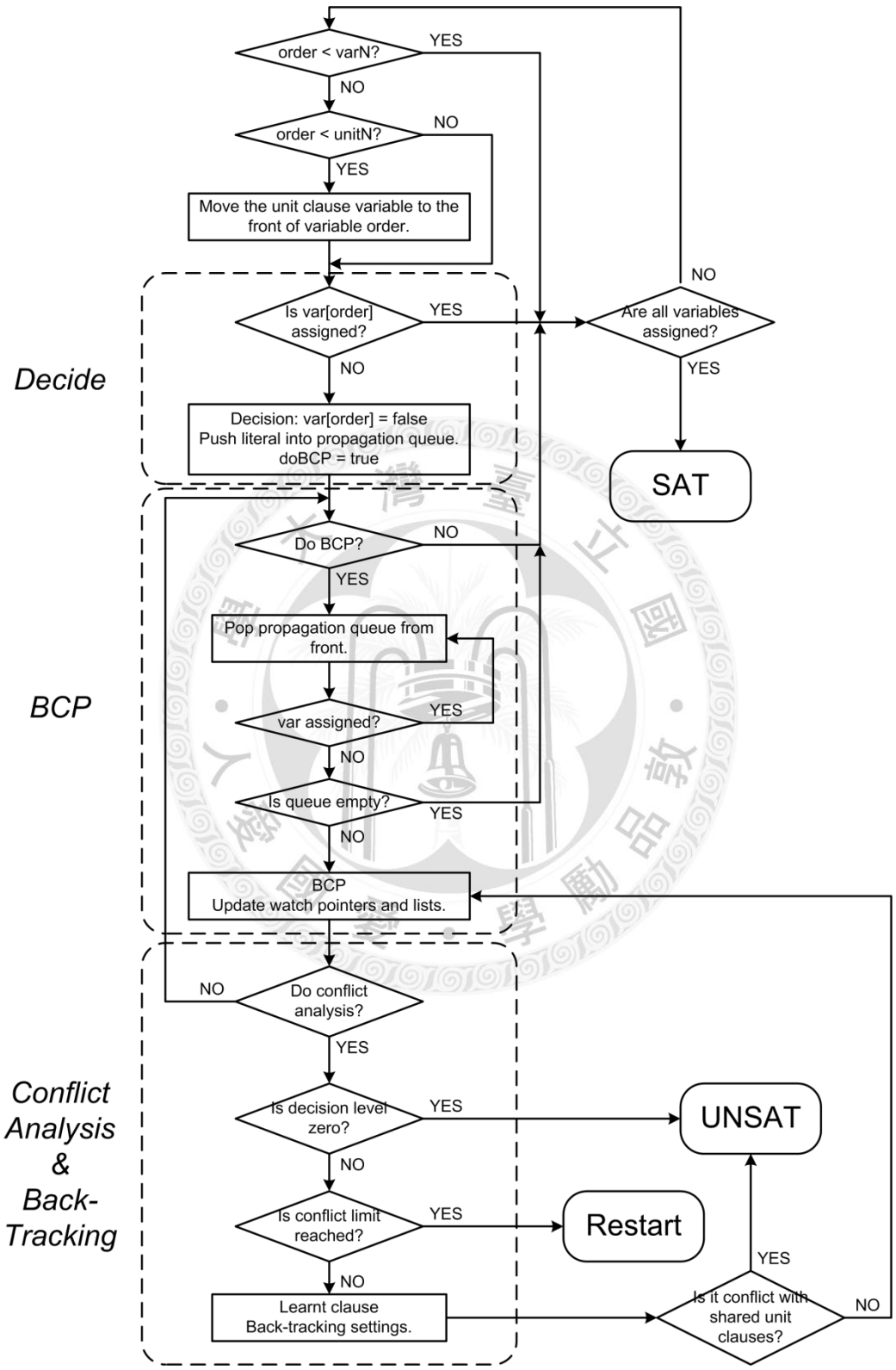


Figure 4.2 - Program flow of CUDASAT solver phase.

order: It represents the *i*-th variable in variable ordering. Each solver has a different pre-determined variable ordering by random shuffling.

varN: The total number of variables.

unitN: The number of current known unit assumptions. The list of unit assumptions is updated in restart phase. We keep all unit assumptions at the front of variable ordering.

var[order]: The variable assignment at the *order*-th variable.

Other terms will be described in following sections.

- *Unit clause* has two meanings. In CUDASAT solver phase, *unit clauses* are the clauses with only one literal. Thus these literals are constants for the problem to be satisfiable. In *Boolean constraint propagation* (BCP), a *unit clause* is a clause with only one unassigned literal. To make that clause to be satisfiable, this literal must be assigned to true.

The goal of a SAT solver is to find a set of variable assignment that makes the problem satisfied. If no such a variable assignment exists, the problem is unsatisfiable.

4.3.1 DPLL (Davis-Putnam-Logemann-Loveland) Algorithm

DPLL algorithm is a systematic search in a satisfiability problem's solution space. Its process could be represented by a binary decision tree. The depth of tree is the number of variables. Since the goal is find a set of variable assignment that satisfies the problem, it could be viewed as to find a leaf node that answers the question. It is a depth-first-search style for exploring the solution space.

The modified DPLL-CDCL algorithm used in MiniSAT [26] is described blow.

	DPLL-CDCL search
1	**loop**
2	*propagate*() // propagate unit clauses.
3	**if** not conflict
4	**if** all variables are assigned
5	**return** SATISFIABLE
6	**else** *decide*() // pick a new variable and assign it.
7	**else**
8	*analyze*() // analyze this conflict and add a learnt clause.
9	**if** top-level conflict
10	**return** UNSATISFIABLE
11	**else**
12	*backtrack*() // undo assignments until conflict is unit.

Table 4.1 - Modified DPLL-CDCL algorithm in MiniSAT

propagate(): performs *Boolean constraint propagation* (BCP). It returns false when a conflict is encountered. BCP is described in section 4.3.2.

decide(): selects an unassigned variable and assign it to true or false.

analyze(): analyzes the source of conflict, then add a conflict clause to represent a learnt constraint. This learnt clause prevents the future decisions from making the same mistake (conflict). Conflict analysis and related techniques are described in section 4.3.4 to 4.3.6.

backtrack(): undoes assignments until the conflict clause is unit. This technique is referred to *non-chronological backtracking* [3]. Back-tracking is described in section 4.3.7.

The implementation of DPLL-CDCL solver is essentially the same in CUDASAT, but with much difference in CUDA oriented code.

4.3.2 Boolean Constraint Propagation (BCP)

BCP is based on the *unit clause rule* [1]: if a clause is unit, its unassigned literal must be true for this clause to be satisfied. The iterated application of unit clause rule is referred to as *unit propagation* or *Boolean constraint propagation* (BCP) [38].

In the search process of DPLL with some variables are assigned, there are 4 states for a clause:

(1) *Unresolved*: two or more than two literals in this clause are unassigned. Thus it is not satisfied, unsatisfied, or unit.
(2) *Unit*: Only one literal is unassigned in a clause. To make this clause to be satisfied, that unassigned literal must be set to true. This literal is *implied* and be added in the *implication queue*.
(3) *Satisfied*: At least one literal in this clause is true.
(4) *Unsatisfied*: All literals in this clause are false.

Unit propagation is applied after each decision of variable assignment. An assignment may make some clauses to be unit clauses. From these unit clauses, it will derive new assignments for subsequent assignments. It may cause some clauses shift from unresolved state to unit, satisfied, or unsatisfied. If any unsatisfied clause is identified, it is called a *conflict*. Then the reason of conflict must be analyzed, and back-track to the lowest level of variable assignment without conflict.

A FIFO (First-In, First-Out) style *implication queue* is used to manage the implied

literals. By popping one implied literal once a time, BCP will continue before it encounters a conflict or an empty implication queue.

Consider an example satisfiability problem *P* in the form of CNF:

$$P := \left(V_1 + \overline{V_2}\right)_{C_1} \left(\overline{V_1} + V_3\right)_{C_2} \left(V_2 + \overline{V_3}\right)_{C_3}$$

A possible BCP process may be:

(1) Making decision: V_1 = false.

(2) Checking the clauses involving V_1: for clause C_1, $\overline{V_2}$ is the only unassigned literal while C_1 is not yet satisfied. Thus $\overline{V_2}$ is added into the implication queue *Q*. $Q = \left\{\overline{V_2}\right\}$. In C_2, V_1 = false makes C_2 to be true, thus C_2 is already satisfied.

(3) *Q* is not empty: $\overline{V_2}$ is popped and it must be true by implication. Thus V_2 = false.

(4) Checking the clauses involving V_2: C_1 is already true. C_3 is not yet satisfied. Thus $\overline{V_3}$ is added into Q. $Q = \left\{\overline{V_3}\right\}$.

(5) *Q* is not empty: $\overline{V_3}$ is popped and it must be true by implication. Thus V_3 = false.

(6) Checking the clauses involving V_3: C_2 is already satisfied. C_3 is also satisfied.

(7) No conflict is encountered. *Q* is empty. BCP stops.

At the end, V_1, V_2, and V_3 are all assigned. C_1, C_2, and C_3 are all satisfied. Thus *P* is

satisfiable with one set of variable assignment $\{V_1 = false, V_2 = false, V_3 = false\}$. Please note that there exists another set of variable assignment for *P* to be satisfied, e.g. $\{V_1 = true, V_2 = true, V_3 = true\}$. The former solution is found by one decision and two propagations.

By an empirical analysis in [7], a solver spends almost 90% of time in BCP. Therefore BCP is the most critical part which needs optimization.

4.3.3 Two-Watched-Literal Scheme

To speed-up BCP, [7] proposed a *two-watched-literal scheme*. It greatly reduced the literals and clauses needed to be checked during BCP. In this scheme, every clause has only two literal to be *watched literals*. These watch literals must be assigned to be true or unassigned. If any of them is false, a false literal is replaced with an unassigned or true literal. There are 6 situations for two watched literals in one clause:

(1) Unassigned and unassigned: it keeps watching these literals.

(2) Unassigned and true: this clause is satisfied. As long as there is at least one true literal, it keeps watching them.

(3) Unassigned and false: the false literal must be replaced with another unassigned or true literal. If no replacement could be found, it indicates all but one (watched) literal is false in this clause. Thus this unassigned literal must be true for this clause to be satisfied. The unassigned literal is added into implication queue.

(4) True and true: this clause is already satisfied.

(5) True and false: this clause is already satisfied.

(6) False and false: it means all other literals in this clause are all false. Thus this clause is unsatisfied and indicates a *conflict*. Subsequent conflict analysis and back-tracking are needed.

Consider an example clause *C* in a satisfiability problem:

$$\left(V_1^{\,u} + \overline{V_2^{\,u}} + \overline{v_3^{\,u}} + v_4^{\,u}\right)$$

Capitalized literals are watched literals. Superscripts are assignments.

A process of two-watched-literal scheme may be:

(1) v_4 is assigned to be false by decision or implication in BCP. Since it is not being watched, clause *C* does not appear in v_4's watch list. BCP does not check this clause. Now clause *C* looks like: $\left(V_1^{\,u} + \overline{V_2^{\,u}} + \overline{v_3^{\,u}} + v_4^{\,F}\right)$.

(2) v_2 is assigned to be true by decision or implication in BCP. Clause *C* is on v_2's watch list while it has the watch literal v_2. It tries to replace v_2 with another non-false literal and finds v_3. Now clause *C* looks like: $\left(V_1^{\,u} + \overline{v_2^{\,T}} + \overline{V_3^{\,u}} + v_4^{\,F}\right)$.

(3) v_1 is assigned to be false by decision or implication in BCP. Clause *C* is on v_1's watch list while it has the watch literal v_1. It tries to replace v_1 with another non-false literal and finds no one. Thus v_3 is false by implication. Now clause *C* looks like: $\left(V_1^{\,F} + \overline{v_2^{\,T}} + \overline{V_3^{\,f}} + v_4^{\,F}\right)$. f means this assignment is an implication

waiting in the implication queue to be propagated.

(4) The benefit of two-watched-literal scheme lies in back-tracking. If v_1's assignment is undone in back-tracking, it is still a watched literal. Because a conflict indicates that all implications in the same decision level are invalid, v_3 is also undone in back-tracking. Thus v_1 and v_3 are still valid watched literals for they are unassigned. No further checks and actions are required. Now clause *C* looks like: $\left(V_1^{\,u} + \overline{v_2^{\,T}} + \overline{V_3^{\,u}} + v_4^{\,F}\right)$.

The two-watched-literal scheme provides a base of efficient yet complex data structure for BCP and back-tracking. Especially it updates nothing in back-tracking.

4.3.4 Conflict Analysis

The goal of conflict analysis is to learn a new clause that prevents subsequent decisions from the same reason of this conflict. It is based on an implication graph that records all decisions and implications from the very first to the conflict point.

An implication graph is a directed acyclic graph (DAG). Each vertex represents a variable assignment. These vertices are usually aligned with their own decision levels. A decision is the root vertex in a decision level. Any implication made by this decision is aligned after the root vertex in the same decision level. A directed edge represents an implication. If an assignment A implies another assignment B, A is the *antecedent* of B. Because of an implication occurs only once for one clause, all vertices except root ones could be viewed as a clause. An antecedent vertex is also called an *antecedent clause*.

For a Boolean satisfiability problem P:

$$P := \left(\overline{v_1} + \overline{v_2}\right)_{C_1} \left(\overline{v_1} + v_3\right)_{C_2} \left(\overline{v_4} + \overline{v_5}\right)_{C_3} \left(v_5 + v_6\right)_{C_4}$$
$$\left(v_7 + \overline{v_8}\right)_{C_5} \left(v_8 + v_9\right)_{C_6} \left(\overline{v_{10}} + v_{11}\right)_{C_7}$$
$$\left(v_2 + \overline{v_3} + v_{12}\right)_{C_8} \left(\overline{v_1} + v_{13} + \overline{v_{14}}\right)_{C_9} \left(v_2 + v_4 + v_{13}\right)_{C_{10}}$$
$$\left(v_8 + v_{10} + v_{14}\right)_{C_{11}} \left(v_5 + \overline{v_9} + \overline{v_{15}}\right)_{C_{12}} \left(\overline{v_6} + \overline{v_{11}} + v_{15}\right)_{C_{13}}$$

An implication graph may look like this:

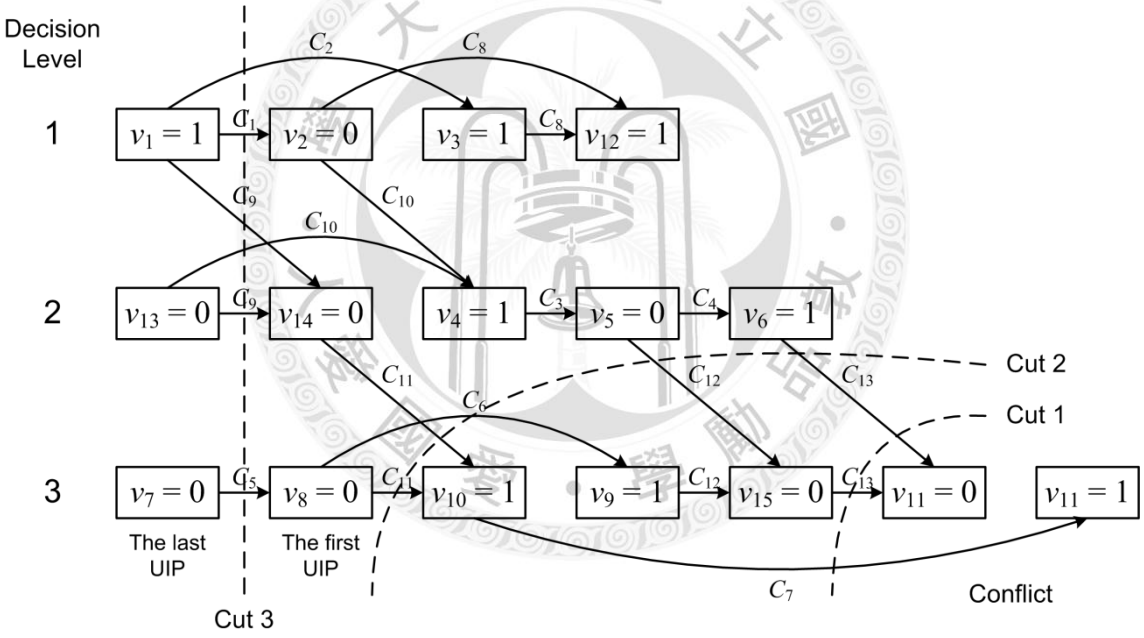


Figure 4.3 - Implication graph and the unique implication point (UIP)

The first decision is $v_1 = 1$, which implies $v_2 = 0$ by C_1 and $v_3 = 1$ by C_2 immediately. By the principle of unit clause rule in BCP, this decision implied $v_2 = 0$, $v_3 = 1$, and $v_{12} = 1$ in total. These assignments are of decision level 1. The root vertex of decision level 1 is $v_1 = 1$. Subsequent decisions and propagations finally encounter a

conflict at v_{11}. Now we need to analyze the reason of conflict.

As the implication graph recorded, each implication could be traced back to which clause implies it. Thus every assigned variable have one *antecedent clause* except those assigned by decision at root vertices. For example, the antecedent clause of v_{15} is C_{12}.

To trace the origin of conflict at v_{11}, we need to trace the antecedent clauses of v_{11} = 0 and v_{11} = 1 simultaneously. We need to find all paths that points to both v_{11} = 0 and v_{11} = 1. We could identify cuts that all implication paths points to the conflict must pass them. For an example, cut 1 could identify C_7 and C_{13} as the origin of conflict. By the same principle, we could identify more cuts like cut 2 and cut 3 in Figure 3.3 that are also the origins of conflict.

All cut represent the same conflict reason by different set of clauses. While we only need one cut for clause learning in section 4.3.5. A heuristic picking the most useful cut is introduced in section 4.3.6.

4.3.5 Clause Learning

The clause learning technique is first described in [3]. It is a sequence of selective resolution operations [39] [40]. For cut 1, $\left(\overline{v_{10}} + v_{11}\right)_{C_7}$ and $\left(\overline{v_6} + \overline{v_{11}} + v_{15}\right)_{C_{13}}$ are the source of conflict at v_{11}. These assignments are v_6 = 1, v_{10} = 1, and v_{15} = 0. To prevent these assignments from future decision again, we could add a new constraint like those constraints (in the form of clauses) into the original problem. The reason of conflict here

is $\left(v_6 \wedge v_{10} \wedge \overline{v_{15}}\right)$. Either C_7 or C_{13} is always unsatisfied by this condition. By De Morgan's law, the inverse logic is $\left(\overline{v_6} + \overline{v_{10}} + v_{15}\right)$. If we add this constraint in the original problem, it guarantees v_6 = 1, v_{10} = 1, and v_{15} = 0 will never happen in the future. This is a learnt clause from conflict analysis. While there are multiple cuts exist in the implication graph, we could learn a new clause $\left(v_5 + \overline{v_6} + v_8 + v_{14}\right)$ by cut 2 and another clause $\left(\overline{v_1} + v_7 + v_{13}\right)$ by cut 3. Please note there are more cuts exist in Figure 3.3. Each learnt clause could prevent the same conflict at v_{11}. Therefore we only need to add one of them into the original problem. But which one is better?

4.3.6 The First Unique Implication Point (UIP)

The *unique implication point* (UIP) [3] is a dominator vertex in the implication graph at the last decision level *l*. In a directed graph *G*, if vertex *d* lies on every path from vertex *u* to vertex v, then *d* is called a *dominator* of *v* [41]. UIP could be identified in linear time [42] in an implication graph. Each UIP could be viewed as a trigger point of implications that leads to the same conflict. There are two UIPs, v_8 = 0 and v_7 = 0 in Figure 3.3. v_8 = 0 is called the first UIP because it is the first UIP encountered at the last decision level by tracing back paths from conflict at v_{11}. v_7 = 0 is called the last UIP for it is the root vertex at decision level 3.

The first UIP is the most effective one for clause learning by empirical experimental results in [43]. Thus CDCL SAT solvers commonly exercise the first UIP clause learning. We simply choose cut 2 in Figure 3.3 as our learnt clause while it is

easy to find and effective.

4.3.7 Non-Chronological Back-Tracking

In [3], it takes the inverse variable assignment of current decision level after a conflict is encountered. Back-tracking is taken when both assignments of the same variable have conflicts. It is more like a trial-and-error scheme and learns every clause at all UIPs. In [7], back-tracking is always taken immediately after a conflict. It goes back to the max level variable (except current decision level) in the learnt clause. In [43], the results indicate that the first UIP clause learning in Chaff [7] may cause more backtracking than in GRASP [3]. However, Chaff creates fewer clauses and is more effective at back-tracking.

Here is an example of non-chronological back-tracking (modified from [44]):

For a satisfiability problem *P*:

$$\left(\overline{v_1}+v_2+v_3\right)_{C_1}\left(v_1+v_3+v_4\right)_{C_2}\left(v_1+v_3+\overline{v_4}\right)_{C_3}\left(v_1+\overline{v_3}+v_4\right)_{C_4}$$
$$\left(v_1+\overline{v_3}+\overline{v_4}\right)_{C_5}\left(\overline{v_2}+\overline{v_3}+v_4\right)_{C_6}\left(\overline{v_1}+v_2+\overline{v_3}\right)_{C_7}\left(\overline{v_1}+\overline{v_2}+v_3\right)_{C_8}$$

(1) Decision making, level 1: $v_1 = 0$. C_1, C_7, and C_8 are satisfied.

(2) Decision making, level 2: $v_2 = 0$. C_6 is satisfied.

(3) Decision making, level 3: $v_3 = 0$. C_4 and C_5 are satisfied. But C_2 and C_3 are unsatisfied now. The conflict occurs at v_4.

(4) Conflict analysis: we learn a new clause $(v_1 + v_3)_{C_9}$ as described in section 3.4.5. The max level except current level of variables is v_1. It undoes all assignments after v_1. In a common practice of CDCL SAT solvers, it back-tracks to level 1 and take the inverse assignment of UIP, i.e. $v_3 = 1$ as if it is propagating in level 1[†].

(5) Now we have $v_1 = 0$ and $v_3 = 1$, they cause a conflict at C_4 and C_5 at v_4.

(6) Conflict analysis: $v_1 = 0$ is the only UIP at level 1 while $v_3 = 1$ is its implication by C_9. The learnt clause is a unit clause $(v_1)_{C_{10}}$. Thus v_1 must be 1.

(7) The same principles of BCP, clause learning, and back-tracking are applied iteratively until a set of satisfiable assignment is found, or no assignment could be found (unsatisfiable).

[†] Back to Figure 3.3, we have a learnt clause $(v_5 + \overline{v_6} + v_8 + v_{14})$ by cut 2. $v_8 = 0$ is the first UIP and clause learning stops here. Now we have added this constraint clause and need to satisfy it. It is a common practice for CDCL SAT solvers to back-track previous decisions to the max level variable except current decision level 3 in this clause, which is level 2. Since we still keep the decision of level 2, $v_5 = 0$, $v_6 = 1$, and $v_{14} = 0$ are valid implications. It is obvious this clause becomes *unit* (section 3.4.2) and v_8 must be set to 1. In the new implication graph, level 3 is undone and cleared, while $v_8 = 1$ is an implication by this learnt clause after $v_6 = 1$ in level 2. Now we have both variable assignments not causing conflict at v_{11} and a satisfied learnt clause that prevent subsequent decisions from making conflict at v_{11} again.

4.4 Shared Learnt Clauses

In general, each learnt clause from a conflict represents that the corresponding solution space is pruned from future searching. In every conflict, any candidate learnt clause has the same effect to block the conflict from happening again. Each clause represents different pruning solution space since they are different constraints with different literals. The quality of learnt clauses is measured fairly arbitrary. It is hard to predict how subsequent decision path will be taken after a learnt clause is added. Though a learnt clause with the least number of literals may be considered as the best quality for it prunes the largest solution space, future decisions may or may not get benefit from this new constraint.

It is reasonable to say “the more learnt clauses, the better.” in a global sense of searching process. The more pruned space, the less is left for searching. But more learnt clauses result in longer clause list to check in BCP. Since BCP is the most critical bottleneck in solver speed, modern CDCL SAT solvers remove low activity learnt clauses periodically. The activity of a clause is measured by how many times it was involved in conflicts.

Discarding “less-important” information may speed up the solver, but the activity is not the sole quality indicator for learnt clauses. In an unsatisfiable problem, since no solution could be found, all solution space must be pruned before the unsatisfiability is concluded. Loss of information may slow down the searching time in such cases. In general, unsatisfiable problems is hard for DPLL-CDCL SAT solvers. Like the

satisfiability problems, there are also many “hints” in structural problems. Activity may indicate a short-cut that lead a solver to its unsatisfiability core. An unsatisfiability core is the minimum set of clauses that guarantees the unsatisfiability of a given problem. Certainly, learnt clauses are more important in unsatisfiable problems than in satisfiable ones.

In parallel SAT solvers, it is critical to make each solver learn different clauses. ManySAT [19] has several carefully hand-crafted solvers to make them divergent, while plingeling [8] has no clause sharing. The more redundant learnt clauses indicate the more similarity of solver searching space, i.e. similar decision paths. As divergent searching space gives more information of the global structure of a problem, sharing these learnt clauses may help each solver to avoid the space that is already searched by other solvers.

In CUDASAT, every solver takes the shared learnt clauses in restart phases. Thus every solver gets the same constraints after each restart. To guarantee divergence, each solver has a pre-randomized static variable ordering. When the randomization has a good uniform distribution, they are expected to be divergent enough in decision making. The ratio of redundant learnt clauses to total learnt clauses is arbitrary low in our experiment. Searching may converge when several solvers are getting near the same solution in the same solution space.

CUDASAT maintains a clause index capacity with 262144 (256k) clauses including problem and learnt clauses. Because every solver needs to maintain its own watch pointer array for clauses, the memory usage for N solvers and index capacity C is

$C \times 4 + N \times C \times 2 \times 4$. For 256 solvers, it is $262144 \times 4 + 256 \times 262144 \times 2 \times 4 = 537919488$ bytes, 513 MB (each data takes 4 bytes). If we take each clause as 80 bytes† in dynamic memory allocation, 262144 clauses will take at least another 20 MB.

CUDASAT only removes low activity learnt clauses when the index is full.

†Observations in GTX 480 card indicate the minimal device memory allocation unit is 80 bytes.

4.5 Memory Management

CUDA has two types of memory allocation: static and dynamic.

Like C/C++, the static memory allocation on device could be done by declaring an array with static size. CUDA provide another method to allocate static array on device memory from host. In CUDASAT, all static arrays in global memory on device are declared from host before any kernel execution.

CUDASAT relies on dynamic memory allocation heavily for creating/removing learnt clauses. In CUDA 4.0, dynamic device memory allocation in device code like malloc()/free() and new()/delete() are supported. All new memory allocation is forced to be aligned for device memory controller specification. Through observation, the alignment is 80 bytes on GTX 480. Thus any allocation less than 80 bytes are wasted on device. We do not know when and how CUDA recycle the freed or deleted device memory since there is no detailed official documentation on CUDA device memory management. We assume CUDA never recycles these fragmentations.

Memory alignment is a critical issue in CUDA programming. Access of non-aligned address will be extremely slow than aligned address. It is reasonable for good CUDA practice to read a bulk of data which starts from aligned address. But in CUDASAT, since all data are scattered single elements, it suffers a great performance penalty from memory access.

A parameter in CUDA driver controls the maximal size available for dynamic memory allocation. Before kernel execution, CUDASAT sets this parameter to 95% of free memory on device. For example, if we have 500 MB free device memory after static memory allocation, the memory space available for dynamic allocation is 475 MB. After setting this maximal heap size, we could only get memory information as “25 MB is free”. No detailed information inside the heap is available. Thus we use the dynamic space totally blinded. In spite we could predict the maximal memory usage and keep it under the free memory space, it is still often terminated by insufficient device memory. This may indicate a serious memory fragmentation and waste.

As we have most 256 solvers in CUDASAT, each solver gets 6 MB in average on a GTX 480 card with 1536 MB device memory. We also have to reserve certain amount of memory for shared clauses. Therefore we could only run tiny cases for current device memory capacity.

The total runtime includes initialization, solver, and restart time.

Chapter 5 Experimental Results

5.1 Description of Cases

The SAT problems corresponding to real world cases or randomly generated are classified into *industrial* or *random* SAT problems respectively. In general, the UNSAT problems are harder than SAT ones, while random problems are harder than industrial ones of the same size. Thus we have four categories of cases: (1) industrial SAT, (2) industrial UNSAT, (3) random SAT, and (4) random UNSAT problems. We have observed a downward trend in searching events per solver in all test cases. In following sections, we only present the largest executable cases for CUDASAT in each category.

The executable largest case size is about 1000 variables and 10000 clauses. The main restriction is insufficient device memory. Since CUDA dynamic device memory management is not efficient, numerous memory fragments are accumulated after a lot of dynamic allocation/de-allocation. Without memory recycling, no space would be available for memory allocation. Most large cases used up memory space during allocating memory for program data at initialization.

All cases run three configurations described in section 3.3: (1) all-idling, (2) half-idling, and (3) no-idling, labeled as “A”, “H”, and “N” respectively. Each configuration runs 1, 2, 4, 8, 16, 32, 64, 128, and 256 solver instances. Since the results of single solver are virtually the same, the single solver statistics in half-idling (H) and no-idling (N) configuration use the results from all-idling (A).

5.2 Environment Setup

CPU	Intel i7-950 3.06GHz
Host RAM	12 GB DDR3
GPU	NVIDIA GeForce GTX480
Operating System	Microsoft Windows 7
Developing Environment	Microsoft Visual Studio 2008
CUDA Toolkit Version	4.0
CUDA Debugger	NVIDIA Parallel NSight 2.0

Table 5.1 - Experimental environment setup

CUDA cores	480
Processor Clock	1401 MHz
Memory Clock	1848 MHz
Compute Capability	2.0
Constant Memory	64 KB
Global Memory	1471 MB
Shared Memory per Block	48 KB
Multiprocessor (MP)	15
Integer and floating point number cores per MP	32[†]
32-bit registers per block	32768
L1 Cache	16 KB
L2 Cache	768 KB

Table 5.2 - GTX480 CUDA properties

[†] The Compute Visual Profiler reports each multiprocessor is capable for executing only 8 blocks in CUDASAT. Thus at most 120 blocks could be executing concurrently. Any block exceed 120 will be executed in serial. But CUDA will use a very complex execution model to make them look like running concurrently.

5.3 Experimental Results

5.3.1 Industrial SAT Problem: logistics.b

- **logistics.b**: it is from "logistics" set in Planning, SATLIB [45] benchmarks [46]. It has 843 variables and 7301 clauses. Satisfiable.
- logistics.b with all-idling (A) configuration.

	CUDASAT-A									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	1	1	1	1	1	1	0	0	0	2
Conflicts	124	224	427	938	1784	3415	4629	9115	8123	129
Average Conflicts	**124.0**	**112.0**	**106.8**	**117.3**	**111.5**	**106.7**	**72.3**	**71.2**	**31.7**	129
Conflicts Speed (/sec)	0.2	0.3	0.6	1.3	2.0	3.0	49.7	56.5	7.9	5864
Decisions	2342	4333	8909	18524	39622	79579	124812	242071	225673	1423
Average Decisions	2342.0	2166.5	2227.3	2315.5	2476.4	2486.8	1950.2	1891.2	**881.5**	1423
Decisions Speed (/sec)	3.1	6.5	11.8	24.8	44.0	69.2	1340.6	1499.2	219.5	64682
Propagations	17234	25667	51155	109623	208489	391827	515958	1011960	914285	13419
Average Propagations	17234.0	**12833.5**	**12788.8**	13702.9	**13030.6**	**12244.6**	**8061.8**	**7905.9**	**3571.4**	13419
Propagations Speed (/sec)	22.9	38.6	67.6	146.6	231.4	340.7	5541.8	6267.4	889.1	609955
Conflict Literals	3050	5016	7966	16604	30499	53621	60022	114626	96580	815
Average Conflict Literals	3050.0	2508.0	1991.5	2075.5	1906.2	1675.7	937.8	895.5	**377.3**	815
Solver Time (sec)	754.0	665.1	756.6	747.9	901.1	1150.0	93.1	161.5	1028.3	0.022
Restart Time (sec)	0.5	2.3	6.2	18.3	56.3	164.5	0.0	0.0	0.0	N/A
Total Time (sec)	756.1	669.7	766.0	771.1	966.1	1331.7	121.5	301.8	1393.3	0.022
Solver Ratio	0.997	0.993	0.988	0.970	0.933	0.864	0.766	0.535	0.738	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1415	1427	N/A
Heap Size (MB)	1332	1330	1326	1317	1301	1272	1210	1087	842	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	100	197	380	729	1391	2560	0	0	0	N/A

Table 5.3 - logistics.b: statistics with all-idling (A) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

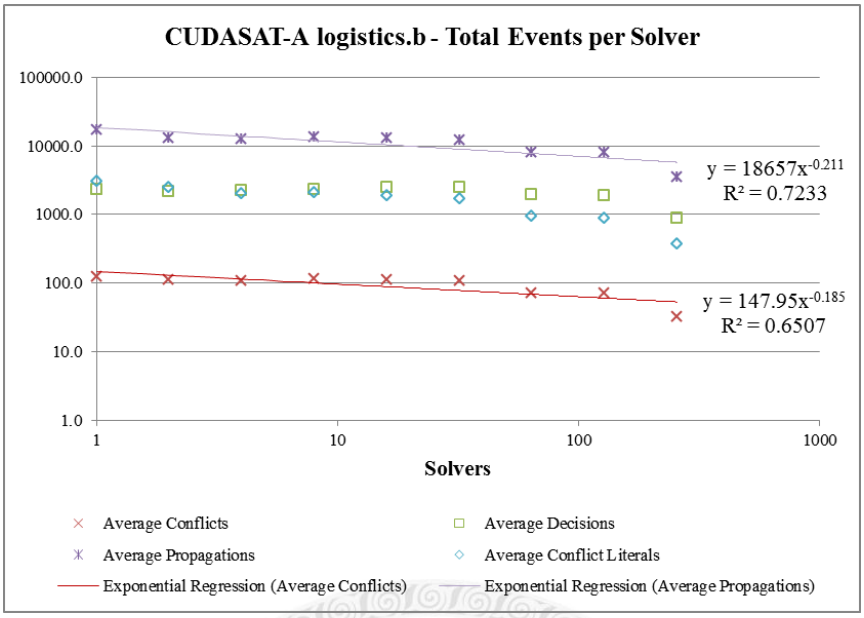


Figure 5.1 - logistics.b: total events per solver with all-idling (A) configuration

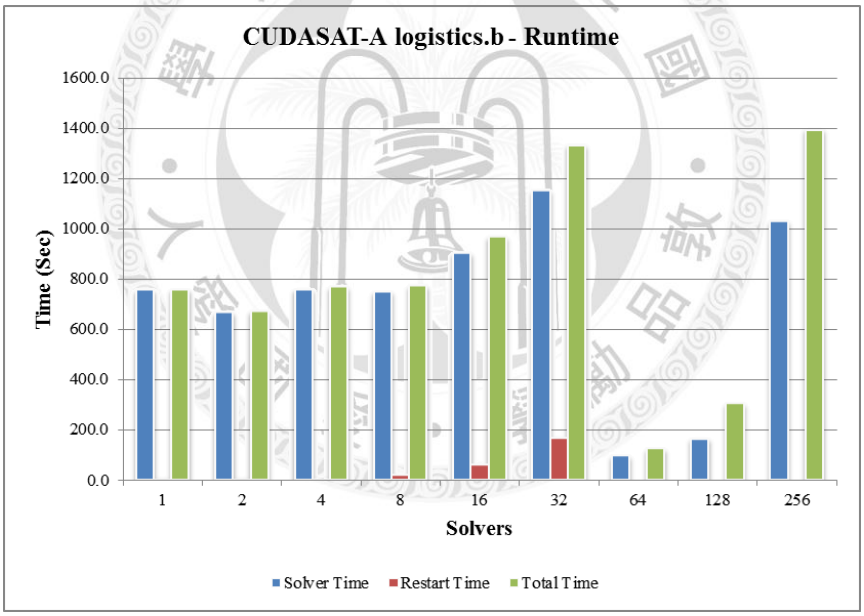


Figure 5.2 - logistics.b: runtime with all-idling (A) configuration

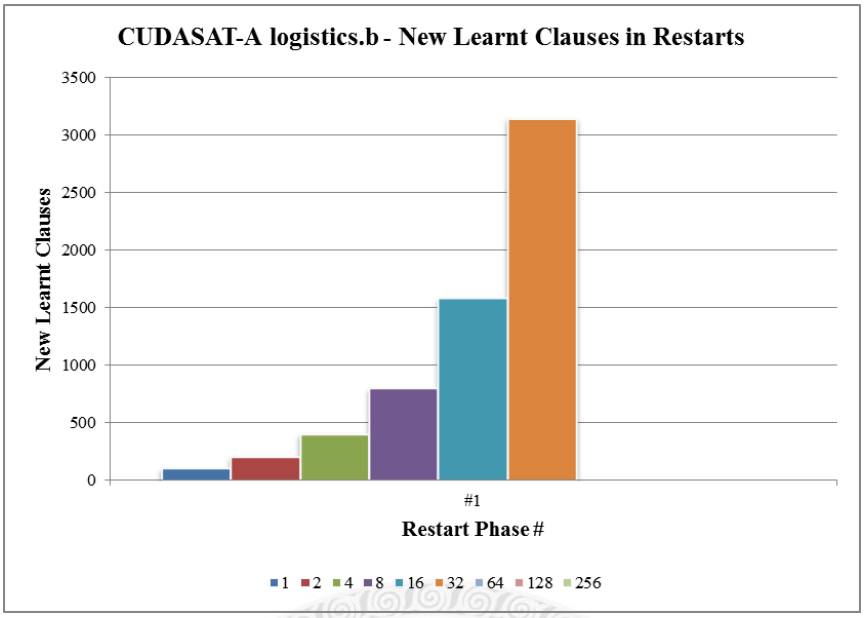


Figure 5.3 - logistics.b: the new learnt clauses in each restart with all-idling (A) configuration

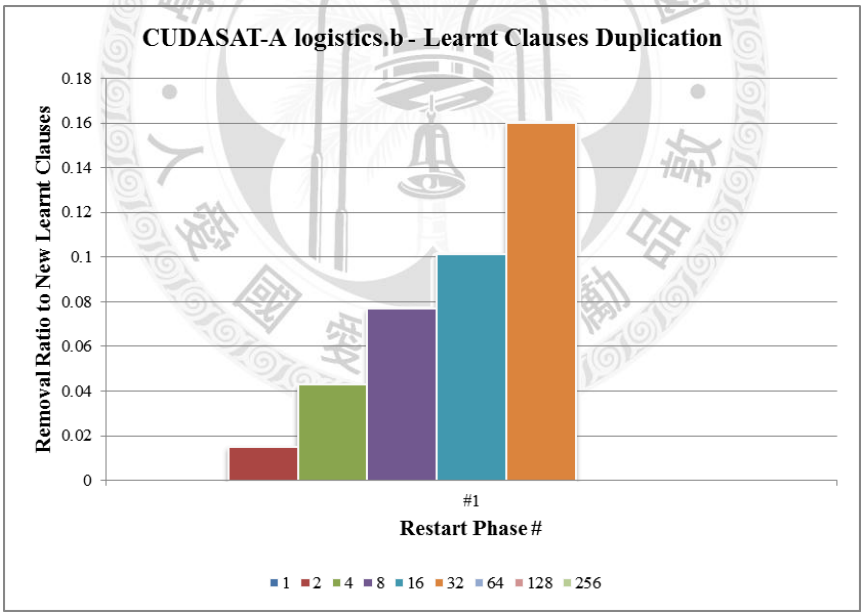


Figure 5.4 - logistics.b: the duplicated ratio in each restart with all-idling (A) configuration

■ logistics.b with half-idling (H) configuration.

	CUDASAT-H									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	1	1	1	1	1	1	0	0	0	2
Conflicts	124	224	374	908	1557	3019	4631	9113	8123	129
Average Conflicts	**124.0**	**112.0**	**93.5**	**113.5**	**97.3**	**94.3**	**72.4**	**71.2**	**31.7**	129
Conflicts Speed (/sec)	0.2	0.3	1.5	3.4	5.3	8.6	49.8	55.8	7.9	5864
Decisions	2342	4333	8443	18133	36151	73246	124961	241942	225688	1423
Average Decisions	2342.0	2166.5	2110.8	2266.6	2259.4	2288.9	1952.5	1890.2	**881.6**	1423
Decisions Speed (/sec)	3.1	6.5	33.5	68.1	123.8	208.8	1343.2	1482.4	219.0	64682
Propagations	17234	25667	44042	102332	178408	345771	516476	1011366	914241	13419
Average Propagations	17234.0	**12833.5**	**11010.5**	**12791.5**	**11150.5**	**10805.3**	**8069.9**	**7901.3**	**3571.3**	13419
Propagations Speed (/sec)	22.9	38.5	175.0	384.2	610.8	985.9	5551.6	6196.9	887.1	609955
Conflict Literals	3050	5016	6440	16276	24754	44452	60038	114627	96580	815
Average Conflict Literals	3050.0	2508.0	1610.0	2034.5	1547.1	1389.1	938.1	895.5	**377.3**	815
Solver Time (sec)	754.0	665.9	251.7	266.3	292.1	350.7	93.0	163.2	1030.6	0.022
Restart Time (sec)	0.5	2.4	5.0	16.6	51.1	143.0	0.0	0.0	0.0	N/A
Total Time (sec)	756.1	670.6	259.7	287.6	351.9	510.7	121.0	303.5	1397.9	0.022
Solver Ratio	0.997	0.993	0.969	0.926	0.830	0.687	0.769	0.538	0.737	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1415	1427	N/A
Heap Size (MB)	1332	1328	1324	1317	1301	1272	1210	1087	842	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	100	187	328	651	1275	2339	0	0	0	N/A

Table 5.4 - logistics.b: statistics with half-idling (H) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

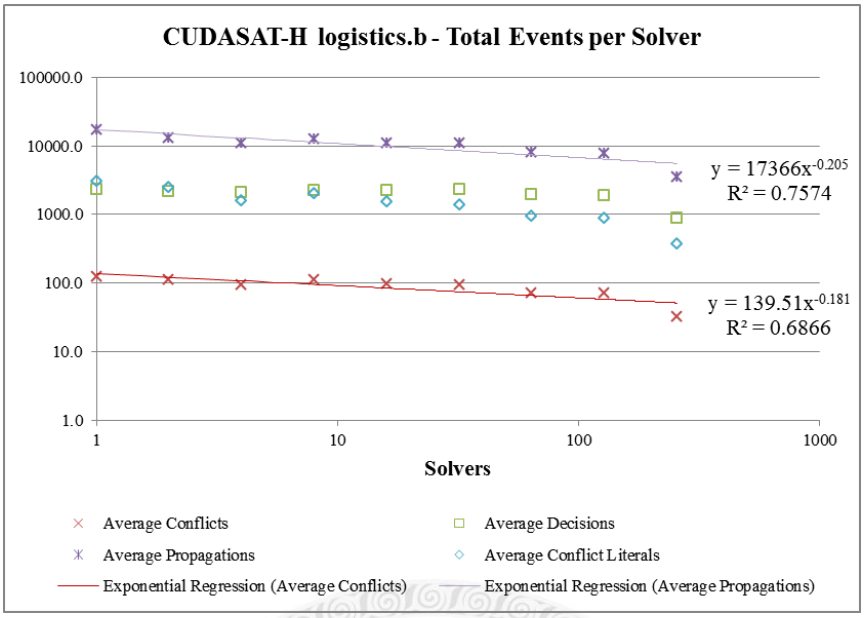


Figure 5.5 - logistics.b: total events per solver of with half-idling (H) configuration

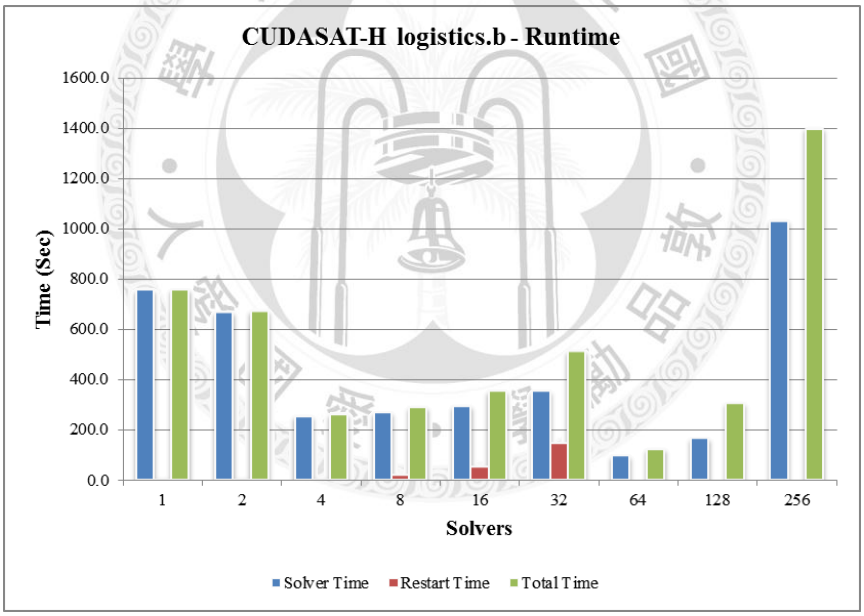


Figure 5.6 - logistics.b: runtime with half-idling (H) configuration

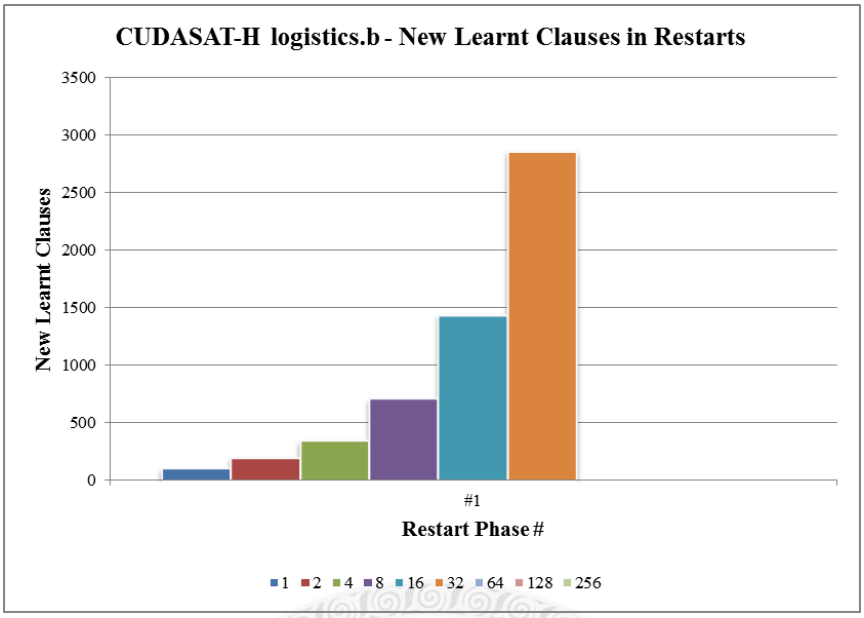


Figure 5.7 - logistics.b: the new learnt clauses in each restart with half-idling (H) configuration

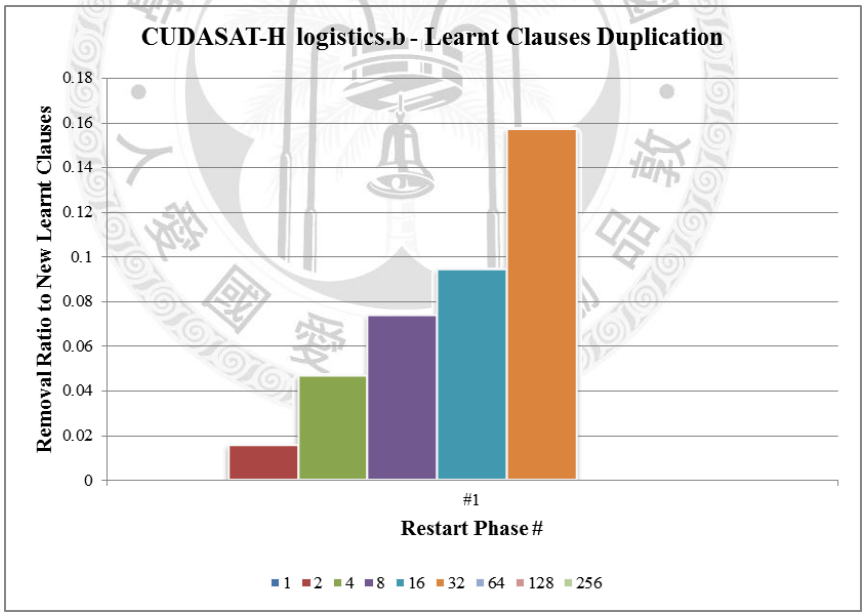


Figure 5.8 - logistics.b: the duplicated ratio in each restart with half-idling (H) configuration

■ logistics.b with no-idling (N) configuration.

	CUDASAT-N									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	1	1	1	1	1	1	1	1	0	2
Conflicts	124	149	327	724	1139	1778	3808	8624	8124	129
Average Conflicts	**124.0**	**74.5**	**81.8**	**90.5**	**71.2**	**55.6**	**59.5**	**67.4**	**31.7**	129
Conflicts Speed (/sec)	0.2	1.3	2.3	5.4	7.4	9.4	7.6	9.5	7.9	5864
Decisions	2342	3390	7553	15479	28982	50648	110649	237440	225780	1423
Average Decisions	2342.0	1695.0	1888.3	1934.9	1811.4	1582.8	1728.9	1855.0	**882.0**	1423
Decisions Speed (/sec)	3.1	29.3	53.4	114.9	188.0	268.6	220.7	261.7	218.9	64682
Propagations	17234	16617	38198	83054	136162	216376	460078	1017253	914886	13419
Average Propagations	17234.0	**8308.5**	**9549.5**	**10381.8**	**8510.1**	**6761.8**	**7188.7**	**7947.3**	**3573.8**	13419
Propagations Speed (/sec)	22.9	143.6	270.0	616.8	883.4	1147.7	917.7	1121.0	886.9	609955
Conflict Literals	3050	3350	5589	12298	16590	21620	47302	106427	96584	815
Average Conflict Literals	3050.0	1675.0	1397.3	1537.3	1036.9	**675.6**	**739.1**	831.5	**377.3**	815
Solver Time (sec)	754.0	115.7	141.5	134.7	154.1	188.5	501.3	907.4	1031.6	0.022
Restart Time (sec)	0.5	1.5	4.0	12.0	37.3	91.1	305.4	1044.8	0.0	N/A
Total Time (sec)	756.1	119.5	148.6	151.4	200.1	295.9	839.9	2121.8	1399.1	0.022
Solver Ratio	0.997	0.968	0.952	0.890	0.770	0.637	0.597	0.428	0.737	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1415	1427	N/A
Heap Size (MB)	1332	1328	1324	1317	1301	1272	1210	1087	842	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	100	128	284	504	936	1355	2704	5563	0	N/A

Table 5.5 - logistics.b: statistics with no-idling (N) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

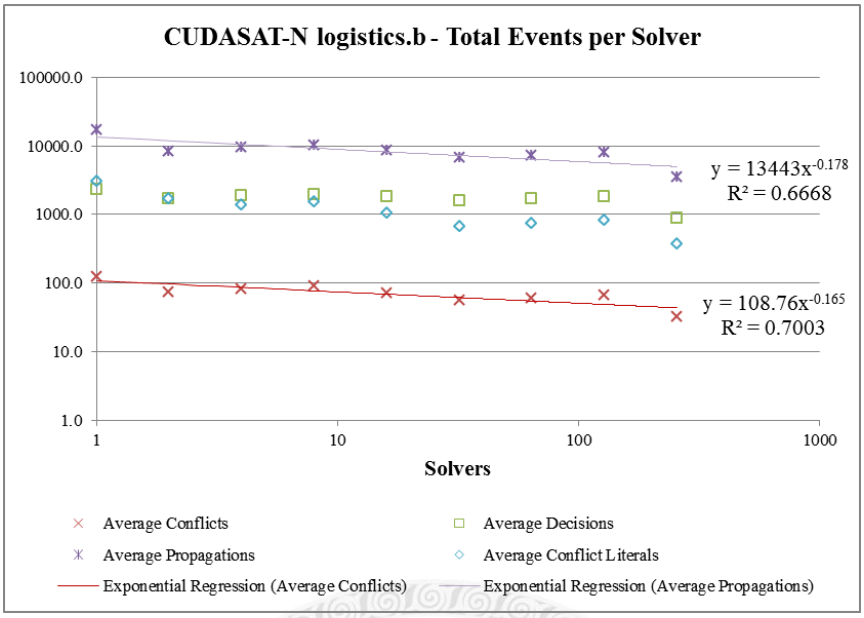


Figure 5.9 - logistics.b: total events per solver of with no-idling (N) configuration

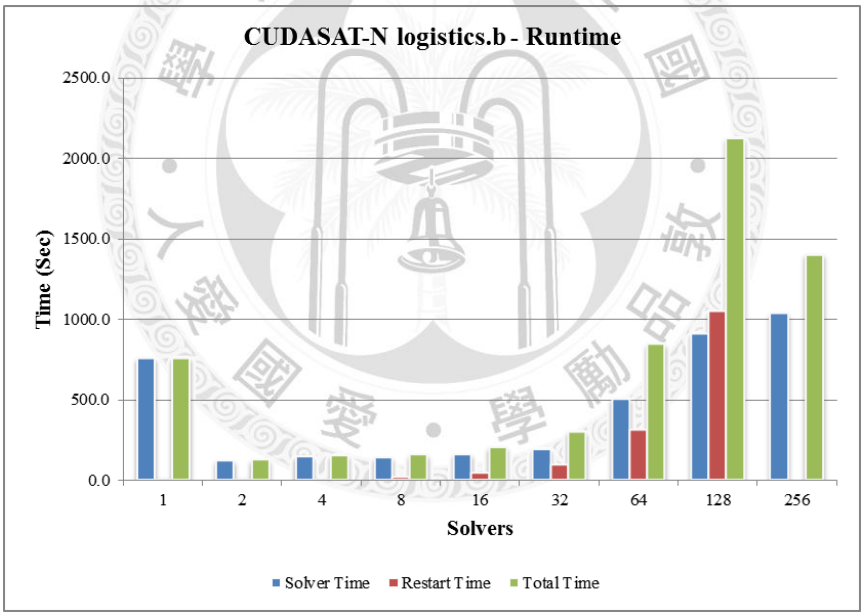


Figure 5.10 - logistics.b: runtime with no-idling (N) configuration

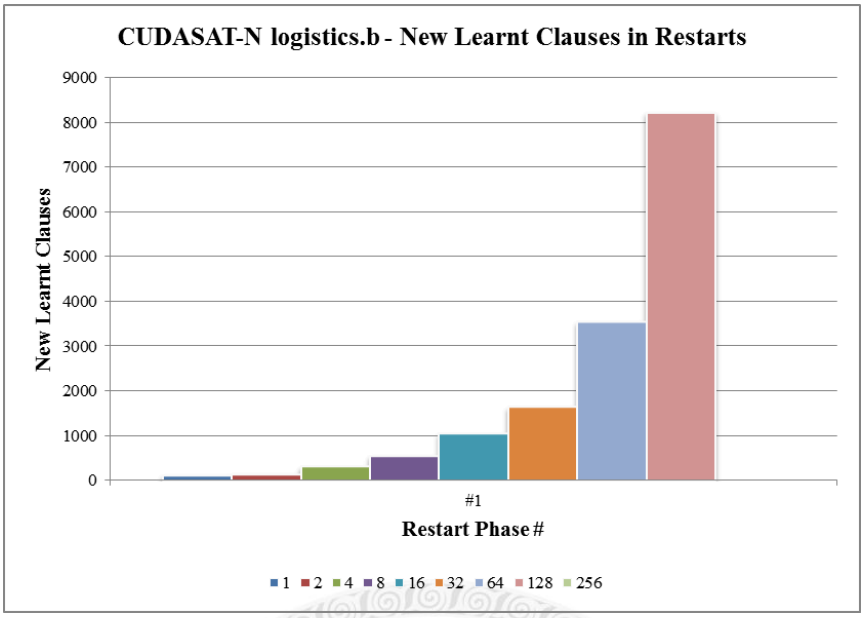


Figure 5.11 - logistics.b: the new learnt clauses in each restart with no-idling (N) configuration

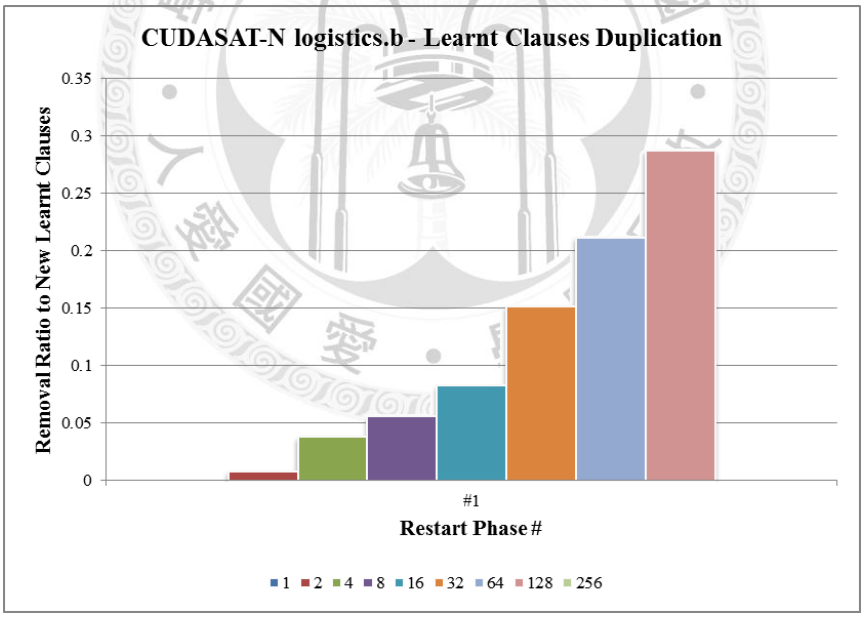


Figure 5.12 - logistics.b: the duplicated ratio in each restart with no-idling (N) configuration

5.3.2 Comparison of Configurations: logistics.b

Figure 5.13 - logistics.b: comparison of configuration A, H, and N

5.3.3 Discussion: logistics.b

- Configuration A:

Among 64, 128, and 256 solvers, at least one solver gets the solution within 100 conflicts and signals a termination of the first solver phase. All other solvers waste their efforts. In this case, the decreasing trend in average events per solver does not reflect the benefit from shared clauses. The ratios of duplicated new learnt clauses are below 16% in all new learnt clauses in the first restart phase. “logistics.b” is an easy-to-SAT case.

The ratio of duplicated new learnt clauses is proportional to the number of solver instances at the first restart phase. This phenomenon is seen in some cases, but not all cases. We will see a dramatically decrease of duplicated learnt clauses in subsequent restart phases. It indicates the learnt clauses are suppressing the duplication.

If all solvers have the same decision order, they will learn exactly the same clauses from the same conflicts. Since 32 solvers have only 16% duplicated learnt clauses in the first restart phase, the divergence of 32 solvers leads a drop of duplication from 100% to 16%.

- Configuration H:

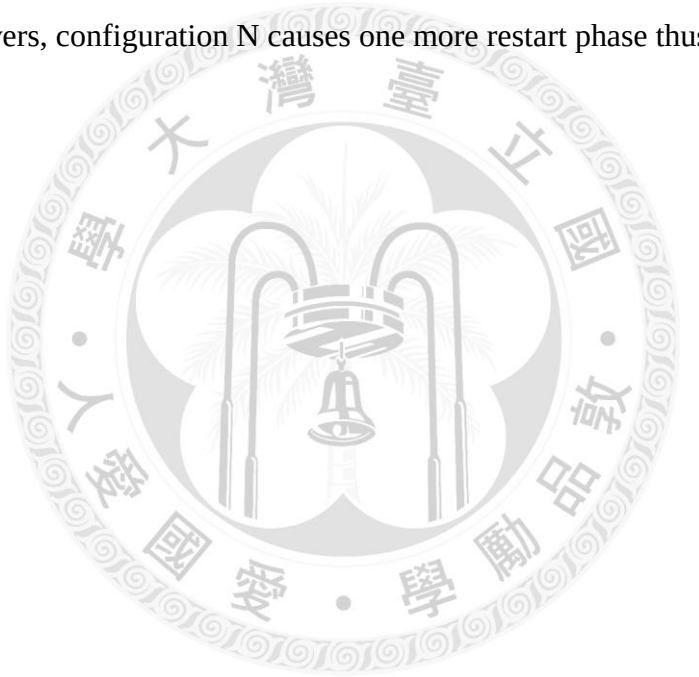
The results are similar to configuration A. The events per solver and runtime of configuration H is significantly lower than configuration A in those cases with a restart phase.

- Configuration N:

The results are similar to configuration H. The events per solver and runtime of configuration N is further lowered in configuration N in those cases with a restart phase.

- Comparison of Configuration A, H, and N

Configuration H and N successfully shortens the events per solver and runtime by preventing solvers from idling at the cost of fewer learnt clauses. In 64 and 128 solvers, configuration N causes one more restart phase thus lengthens the runtime.



5.3.4 Industrial UNSAT Problem: qg4-08

- qg4-08 is from "qg" set in SAT-encoded Quasigroup (or Latin square) instances, SATLIB [45] benchmarks [46]. It has 512 variables and 4984 clauses. Unsatisfiable.
- qg4-08 with all-idling (A) configuration.

	CUDASAT-A									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	5	4	4	4	4	3	3	3	2	6
Conflicts	4407	7103	8401	13821	25008	32988	44553	84877	135387	678
Average Conflicts	4407.0	3551.5	2100.3	1727.6	1563.0	1030.9	696.1	**663.1**	**528.9**	678
Conflicts Speed (/sec)	0.2	0.3	0.6	1.1	1.7	3.4	5.9	4.1	4.9	15767
Decisions	7121	11780	13848	23698	41854	56999	79402	146456	228175	875
Average Decisions	7121.0	5890.0	3462.0	2962.3	2615.9	1781.2	1240.7	1144.2	891.3	875
Decisions Speed (/sec)	0.3	0.6	0.9	1.9	2.9	5.8	10.4	7.0	8.3	20349
Propagations	335012	547992	661149	1067065	1917581	2518683	3433239	6439957	10173780	51343
Average Propagations	335012.0	273996.0	165287.3	133383.1	119848.8	78708.8	53644.4	**50312.2**	**39741.3**	51343
Propagations Speed (/sec)	15.4	26.1	45.2	84.9	131.4	257.8	451.0	309.2	371.1	1194023
Conflict Literals	130769	200357	240482	405957	734798	969184	1302716	2648706	4422261	10417
Average Conflict Literals	130769.0	100178.5	60120.5	50744.6	45924.9	30287.0	20354.9	20693.0	17274.5	10417
Solver Time (sec)	21722.1	20959.3	14615.6	12571.8	14589.1	9768.4	7612.3	20831.0	27412.2	0.043
Restart Time (sec)	16.9	16.0	39.4	95.6	442.9	326.1	832.0	1948.8	2465.7	N/A
Total Time (sec)	21744.4	20984.5	14663.7	12677.5	15051.2	10126.1	8508.0	23138.1	30741.9	0.043
Solver Ratio	0.999	0.999	0.997	0.992	0.969	0.965	0.895	0.900	0.892	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1331	1329	1324	1317	1301	1271	1211	1089	844	N/A
Completed Instances	1	1	1	1	16	1	1	2	1	1
Final Shared Clause	2013	3120	6240	12464	16200	19935	38155	79718	63653	N/A

Table 5.6 - qg4-08: statistics with all-idling (A) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

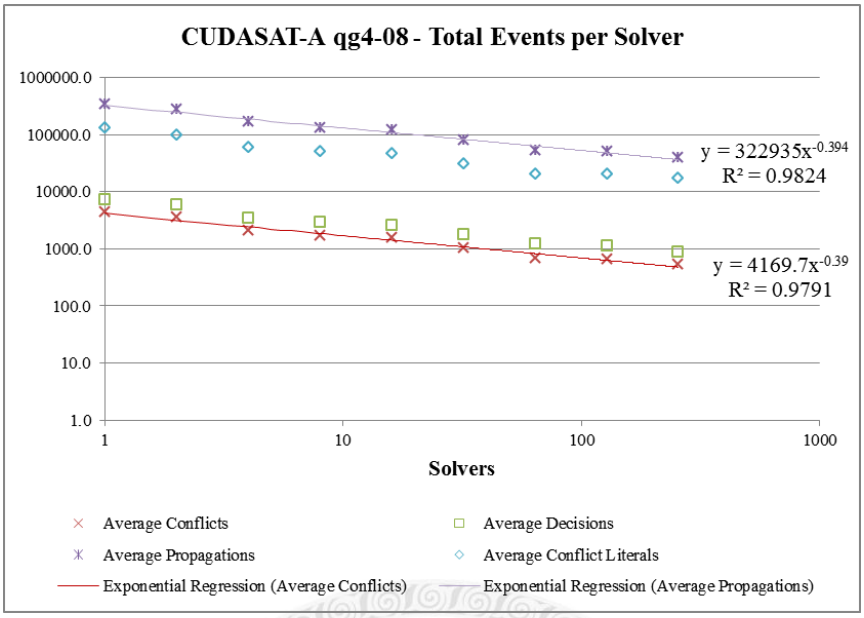


Figure 5.14 - qg4-08: total events per solver of with all-idling (A) configuration

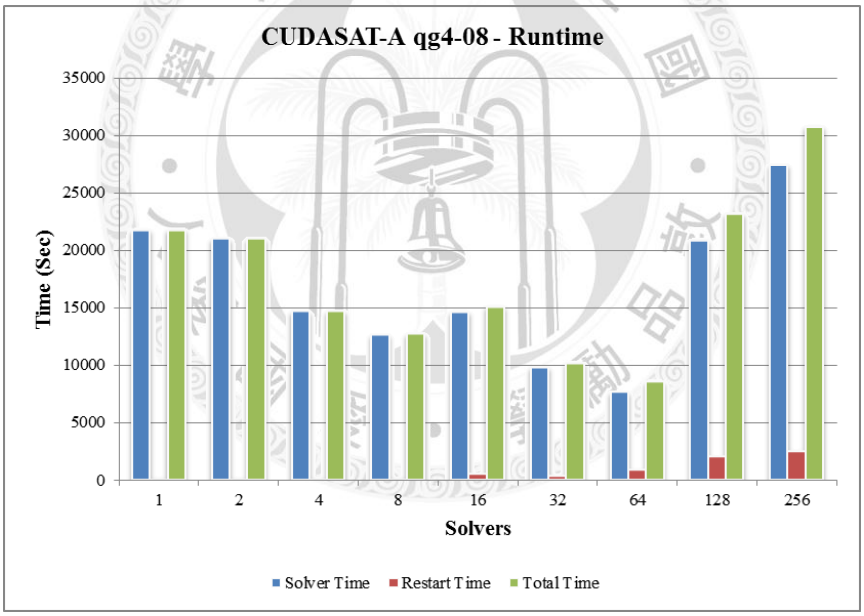


Figure 5.15 - qg4-08: runtime with all-idling (A) configuration

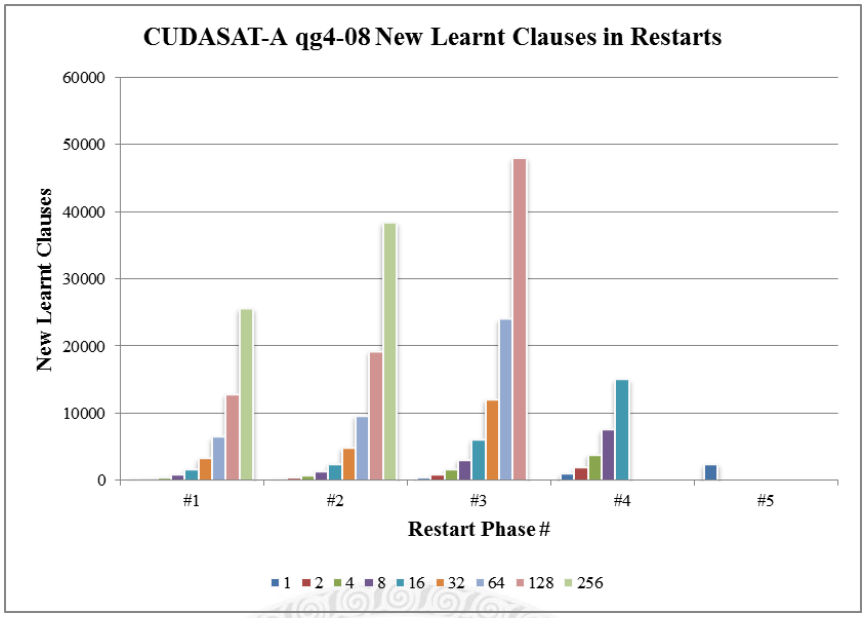


Figure 5.16 - qg4-08: the new learnt clauses in each restart with all-idling (A) configuration

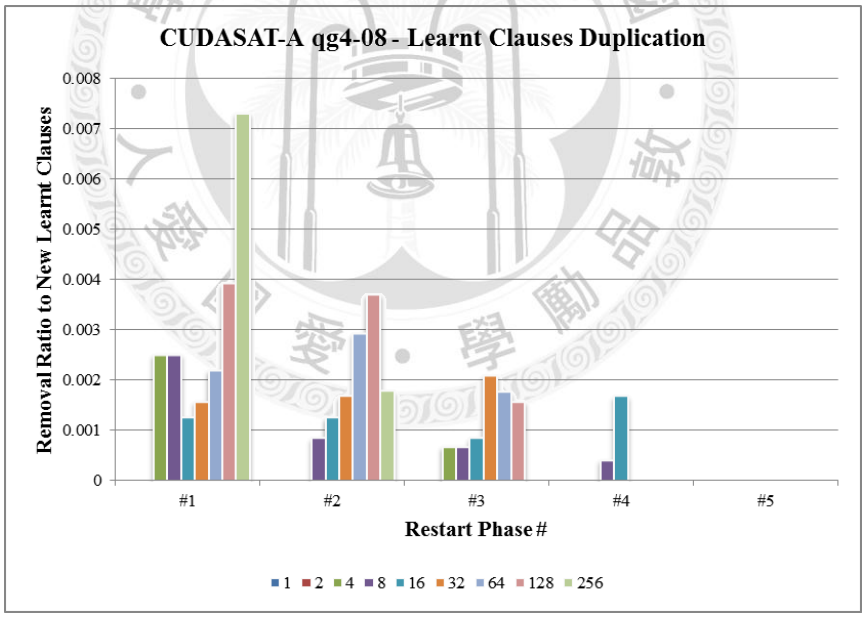


Figure 5.17 - qg4-08: the duplicated ratio in each restart with all-idling (A) configuration

■ qg4-08 with half-idling (H) configuration.

	CUDASAT-H									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	5	4	4	4	4	3	3	3	3	6
Conflicts	4407	7103	7915	14013	24233	37727	44070	77455	136027	678
Average Conflicts	4407.0	3551.5	1978.8	1751.6	1514.6	1179.0	688.6	**605.1**	**531.4**	678
Conflicts Speed (/sec)	0.2	0.3	0.6	1.3	2.6	5.4	8.7	11.2	6.3	15767
Decisions	7121	11780	13353	24257	40608	63497	78696	135942	229340	875
Average Decisions	7121.0	5890.0	3338.3	3032.1	2538.0	1984.3	1229.6	1062.0	895.9	875
Decisions Speed (/sec)	0.3	0.6	1.1	2.3	4.3	9.2	15.5	19.7	10.7	20349
Propagations	335012	547992	619321	1078451	1859840	2856864	3396870	5915372	10279717	51343
Average Propagations	335012.0	273996.0	154830.3	134806.4	116240.0	89277.0	53076.1	**46213.8**	**40155.1**	51343
Propagations Speed (/sec)	15.4	26.2	49.3	100.1	195.9	412.7	668.4	856.4	478.3	1194023
Conflict Literals	130769	200357	224778	410790	707985	1129634	1270506	2389280	4493366	10417
Average Conflict Literals	130769.0	100178.5	56194.5	51348.8	44249.1	35301.1	19851.7	18666.3	17552.2	10417
Solver Time (sec)	21722.1	20930.3	12568.3	10769.3	9493.1	6922.5	5081.8	6907.1	21492.2	0.043
Restart Time (sec)	16.9	16.0	38.3	94.2	435.8	312.0	796.4	1878.7	4633.4	N/A
Total Time (sec)	21744.4	20955.4	12615.4	10874.0	9946.0	7265.0	5942.5	9156.6	27126.9	0.043
Solver Ratio	0.999	0.999	0.996	0.990	0.954	0.953	0.855	0.754	0.792	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1331	1329	1324	1317	1301	1271	1211	1089	844	N/A
Completed Instances	1	1	1	1	16	1	1	1	5	1
Final Shared Clause	2013	3120	6036	12133	13881	19117	37029	74461	134338	N/A

Table 5.7 - qg4-08: statistics with half-idling (H) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

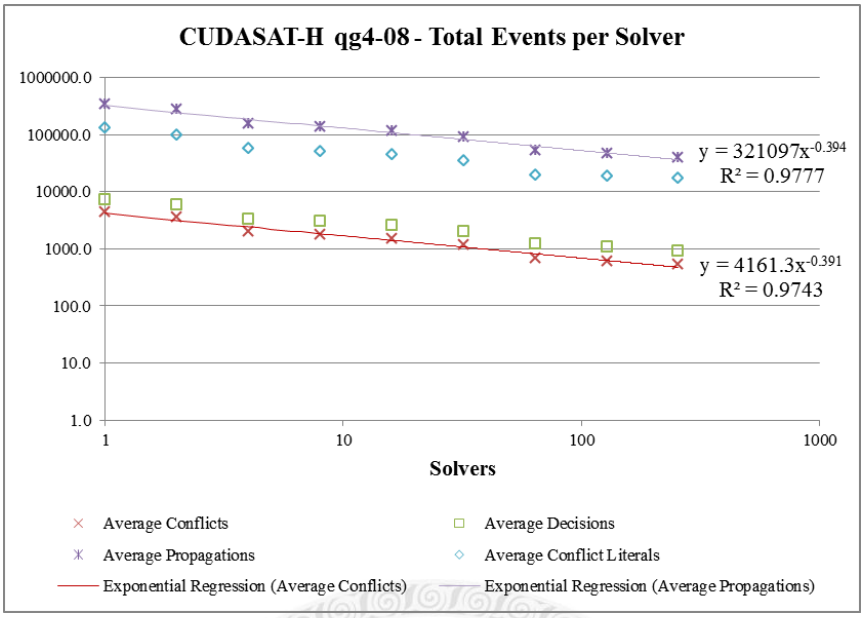


Figure 5.18 - qg4-08: total events per solver of with half-idling (H) configuration

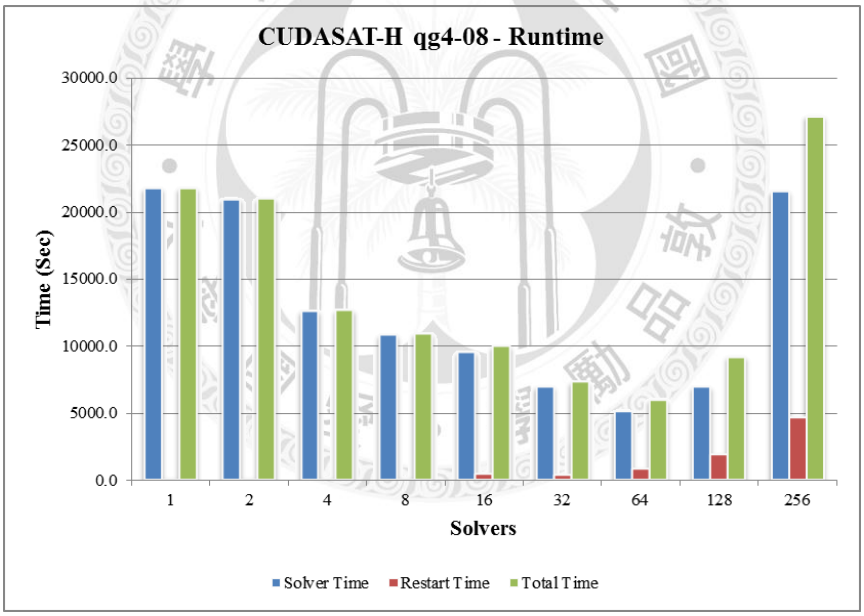


Figure 5.19 - qg4-08: runtime with half-idling (H) configuration

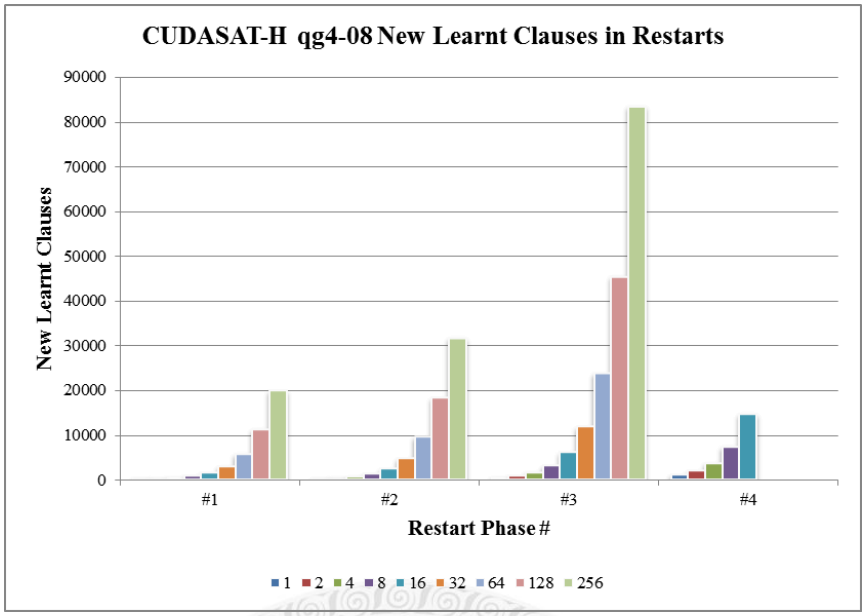


Figure 5.20 - qg4-08: the new learnt clauses in each restart with half-idling (H) configuration

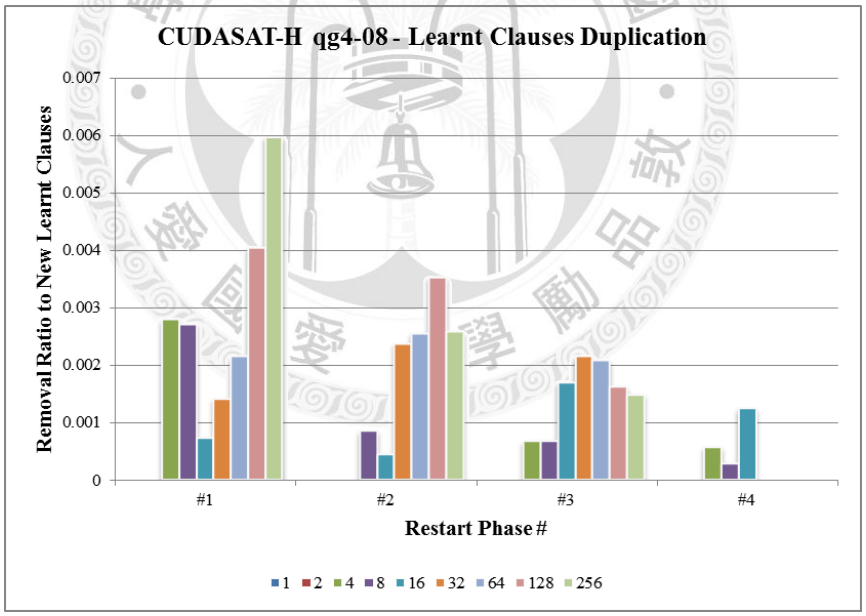


Figure 5.21 - qg4-08: the duplicated ratio in each restart with half-idling (H) configuration

■ qg4-08 with no-idling (N) configuration.

	CUDASAT-N									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	5	4	4	4	4	4	3	3	3	6
Conflicts	4407	6967	7607	15623	19060	36253	52839	74828	75592	678
Average Conflicts	4407.0	3483.5	1901.8	1952.9	1191.3	1132.9	825.6	**584.6**	**295.3**	678
Conflicts Speed (/sec)	0.2	0.4	0.7	1.4	3.0	5.4	8.9	11.7	9.0	15767
Decisions	7121	11500	12860	26093	33012	61299	90720	132494	133521	875
Average Decisions	7121.0	5750.0	3215.0	3261.6	2063.3	1915.6	1417.5	1035.1	**521.6**	875
Decisions Speed (/sec)	0.3	0.6	1.1	2.3	5.2	9.2	15.3	20.7	15.9	20349
Propagations	335012	538587	598138	1190325	1462145	2770043	4034578	5623813	5684803	51343
Average Propagations	335012.0	269293.5	149534.5	148790.6	91384.1	86563.8	63040.3	**43936.0**	**22206.3**	51343
Propagations Speed (/sec)	15.4	28.0	53.2	105.8	229.6	416.4	682.5	879.3	677.6	1194023
Conflict Literals	130769	204919	221426	455845	540043	1079501	1633040	2277876	2309192	10417
Average Conflict Literals	130769.0	102459.5	55356.5	56980.6	33752.7	33734.4	25516.3	17795.9	**9020.3**	10417
Solver Time (sec)	21722.1	19268.8	11249.9	11252.5	6367.3	6652.9	5911.5	6395.7	8389.3	0.043
Restart Time (sec)	16.9	14.7	35.0	84.2	206.2	822.9	642.9	1692.7	5016.4	N/A
Total Time (sec)	21744.4	19292.2	11293.6	11346.3	6590.9	7511.0	6617.8	8375.6	14118.0	0.043
Solver Ratio	0.999	0.999	0.996	0.992	0.966	0.886	0.893	0.764	0.594	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1331	1329	1324	1317	1301	1271	1211	1089	844	N/A
Completed Instances	1	1	1	1	1	32	1	1	1	1
Final Shared Clause	2013	2707	5445	10548	15990	29564	27300	58641	57862	N/A

Table 5.8 - qg4-08: statistics with no-idling (N) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

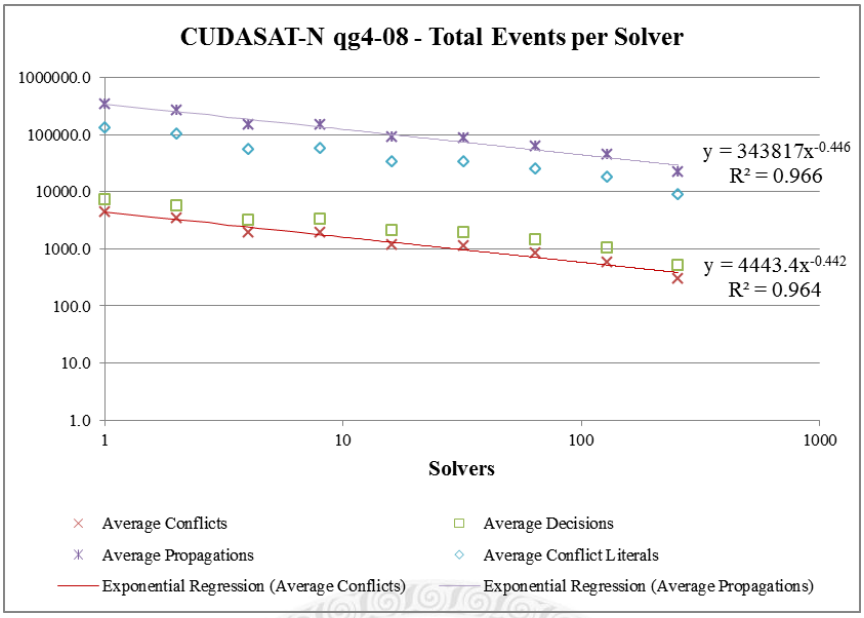


Figure 5.22 - qg4-08: total events per solver of with no-idling (N) configuration

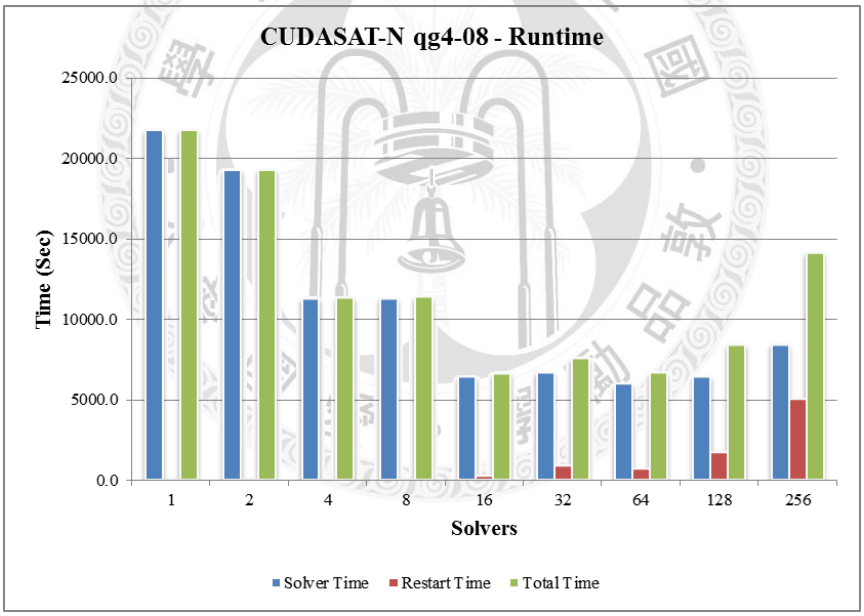


Figure 5.23 - qg4-08: runtime with no-idling (N) configuration

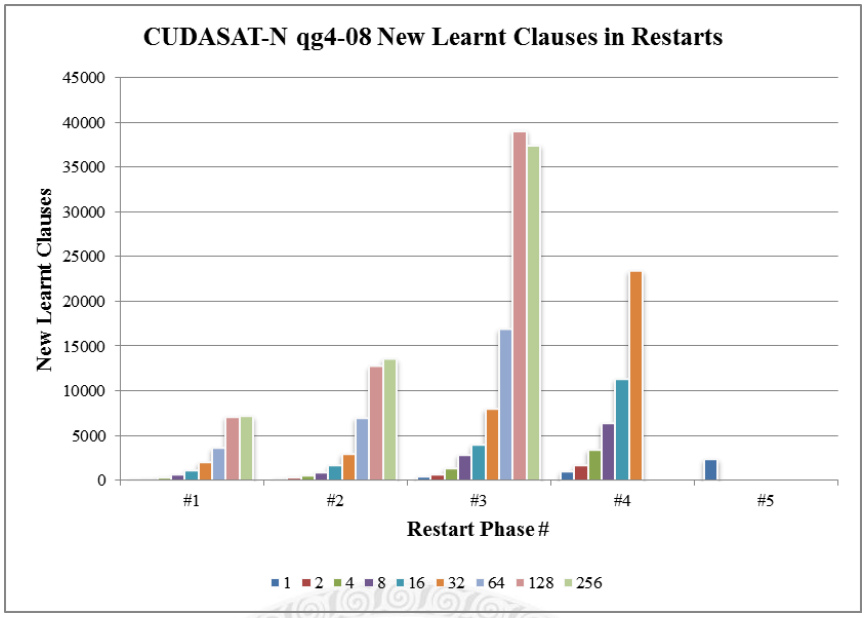


Figure 5.24 - qg4-08: the new learnt clauses in each restart with no-idling (N) configuration

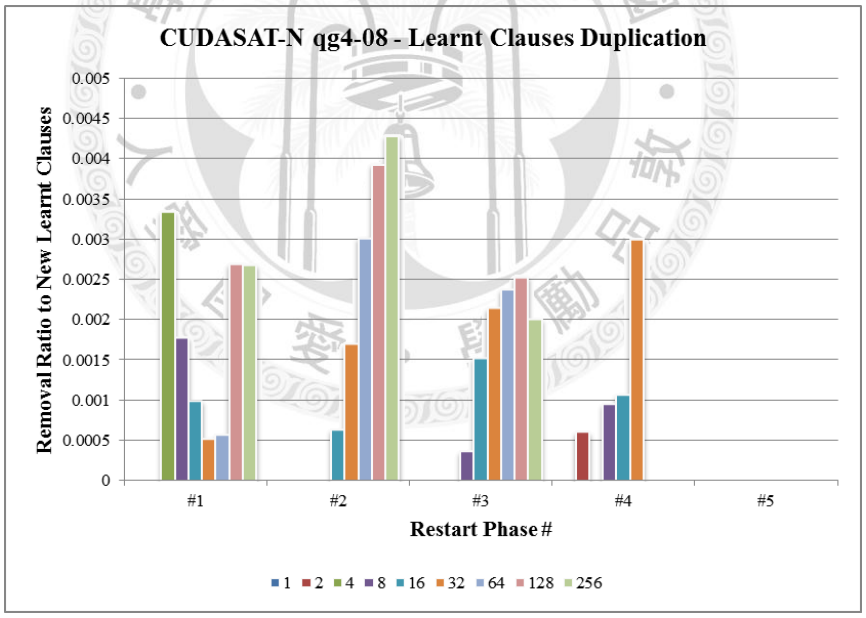


Figure 5.25 - qg4-08: the duplicated ratio in each restart with no-idling (N) configuration

5.3.5 Comparison of Configurations: qg4-08

Figure 5.26 - qg4-08: comparison of configuration A, H, and N

5.3.6 Discussion: qg4-08

- Configuration A

  This case is much harder than logistics.b. The downward trend in conflicts, decision, propagations, and conflict literals is more obvious than those in logistics.b.

  The duplication of learnt clauses is below 0.8%, which is much fewer than in logistics.b. It indicated the behavior of solvers is much divergent in this case.

- Configuration H

  The runtime seems much regular than in configuration A. It indicates that some solvers have much longer search processes in configuration A, making other solvers idling very long. Since we restart when a half of solvers reach their conflict limits, those long search processes could be avoided at the cost of less shared clauses.

  In 256 solvers, the final learnt clauses are more than in configuration A due to one more restart phase.

- Configuration N

  In general, we save more idling solvers in configuration N then in configuration H. But fewer shared clauses are generated in each solver phase and result in more solver phases needed to get the solution for 32 and 256 solvers.

The final learnt clauses are fewer than in configuration H with the same restart phases.

- Comparison of Configuration A, H, and N

  The average events per solver, runtime, and learnt clauses are significantly lower in configuration N. It seems the quality of fewer learnt clauses is still good in configuration N.

5.3.7 Random SAT Problem: f150s

- f150s is a random generated hard-to-SAT problem. It has 150 variables and 640 clauses. Satisfiable.
- f150s with all-idling (A) configuration.

	CUDASAT-A									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	6	6	5	4	3	3	3	3	1	10
Conflicts	20774	41799	21359	16532	12012	25615	41100	85441	50565	1541
Average Conflicts	20774.0	20899.5	5339.8	2066.5	**750.8**	**800.5**	**642.2**	**667.5**	**197.5**	1541
Conflicts Speed (/sec)	2.1	1.4	10.4	26.9	64.6	96.9	120.1	48.1	85.1	57074
Decisions	27620	53123	28378	22003	17516	36441	59783	125567	81875	1864
Average Decisions	27620.0	26561.5	7094.5	2750.4	**1094.8**	**1138.8**	**934.1**	**981.0**	**319.8**	1864
Decisions Speed (/sec)	2.8	1.8	13.8	35.8	94.2	137.9	174.7	70.6	137.8	69037
Propagations	742023	1455117	764052	595462	430796	905226	1440920	2942106	1744223	49636
Average Propagations	742023.0	727558.5	191013.0	74432.8	**26924.8**	**28288.3**	**22514.4**	**22985.2**	**6813.4**	49636
Propagations Speed (/sec)	74.5	49.1	372.6	968.1	2316.2	3426.1	4210.1	1655.2	2936.4	1838370
Conflict Literals	347579	735169	349139	272914	184555	405700	648008	1345999	727718	13762
Average Conflict Literals	347579.0	367584.5	87284.8	34114.3	**11534.7**	**12678.1**	**10125.1**	**10515.6**	**2842.6**	13762
Solver Time (sec)	9965.6	29652.2	2050.6	615.1	186.0	264.2	342.3	1777.5	594.0	0.027
Restart Time (sec)	8.7	32.6	25.6	21.4	18.1	57.5	170.8	508.6	97.2	N/A
Total Time (sec)	9974.5	29691.4	2078.1	637.4	205.0	323.9	519.4	2312.9	714.8	0.027
Solver Ratio	0.999	0.999	0.987	0.965	0.907	0.816	0.659	0.769	0.831	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1272	1211	1089	845	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	9762	19522	15617	12486	9988	19953	39877	79639	24853	N/A

Table 5.9 - f150s: statistics with all-idling (A) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

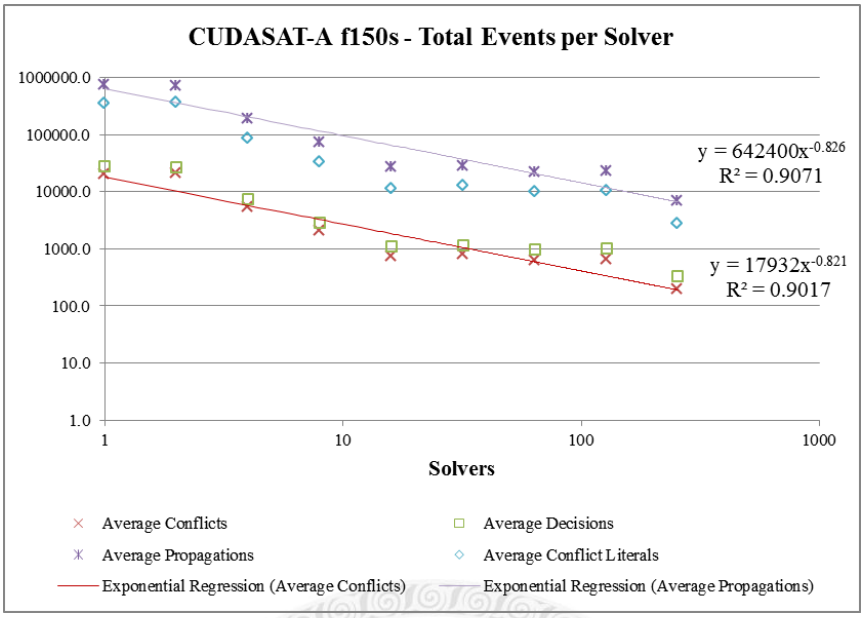


Figure 5.27 - f150s: total events per solver of with all-idling (A) configuration

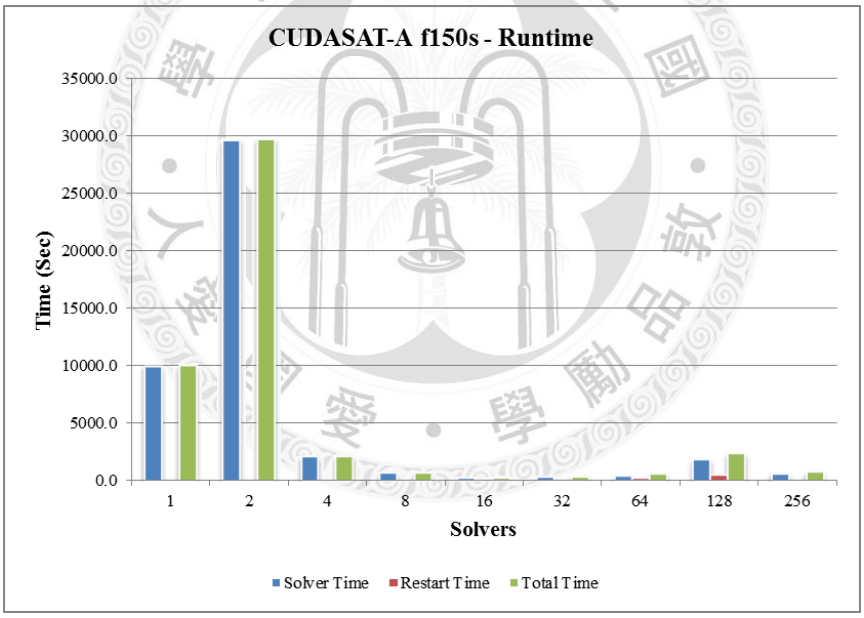


Figure 5.28 - f150s: runtime with all-idling (A) configuration

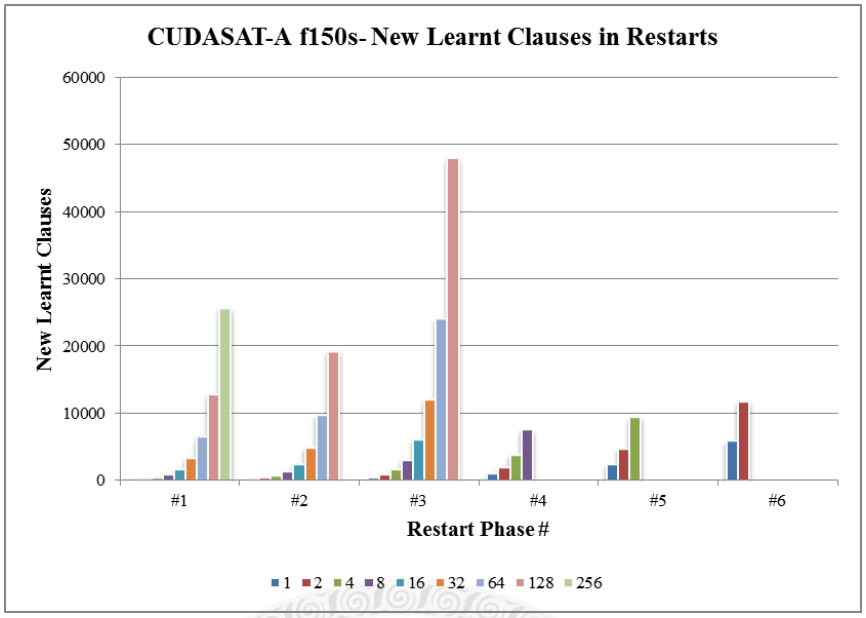


Figure 5.29 – f150s: the new learnt clauses in each restart with all-idling (A) configuration

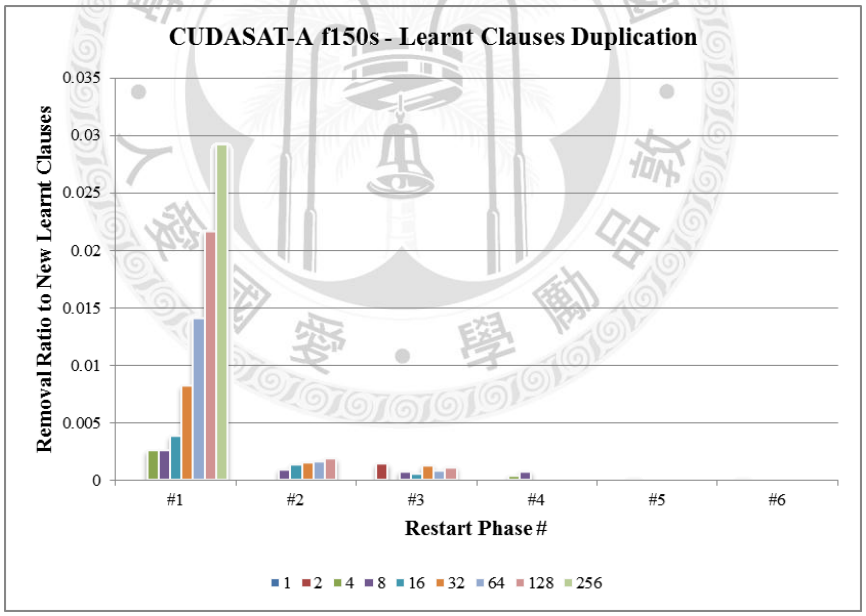


Figure 5.30 – f150s: the duplicated ratio in each restart with all-idling (A) configuration

■ f150s with half-idling (H) configuration.

	CUDASAT-H									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	6	6	5	4	3	3	3	3	1	10
Conflicts	20774	41809	23633	16191	11786	21324	40816	79301	52830	1541
Average Conflicts	20774.0	20904.5	5908.3	2023.9	**736.6**	**666.4**	**637.8**	**619.5**	**206.4**	1541
Conflicts Speed (/sec)	2.1	1.4	11.4	37.7	92.0	142.9	168.2	129.7	92.1	57074
Decisions	27620	53132	31628	21458	17028	30745	59421	117512	84226	1864
Average Decisions	27620.0	26566.0	7907.0	2682.3	**1064.3**	**960.8**	**928.5**	**918.1**	**329.0**	1864
Decisions Speed (/sec)	2.8	1.8	15.2	49.9	132.9	206.1	244.8	192.2	146.9	69037
Propagations	742023	1455556	847538	579408	422983	752310	1433561	2726500	1820440	49636
Average Propagations	742023.0	727778.0	211884.5	72426.0	**26436.4**	**23509.7**	**22399.4**	**21300.8**	**7111.1**	49636
Propagations Speed (/sec)	74.5	48.3	408.3	1347.6	3302.2	5042.5	5906.5	4459.7	3174.5	1838370
Conflict Literals	347579	735386	388633	267745	180647	333403	642695	1244766	767418	13762
Average Conflict Literals	347579.0	367693.0	97158.3	33468.1	**11290.4**	**10418.8**	**10042.1**	**9724.7**	**2997.7**	13762
Solver Time (sec)	9965.6	30162.5	2075.8	430.0	128.1	149.2	242.7	611.4	573.5	0.027
Restart Time (sec)	8.7	33.1	24.1	20.5	17.2	53.7	166.2	463.5	86.5	N/A
Total Time (sec)	9974.5	30202.4	2101.8	451.4	146.2	205.1	414.8	1101.5	678.4	0.027
Solver Ratio	0.999	0.999	0.988	0.953	0.876	0.728	0.585	0.555	0.845	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1272	1211	1089	845	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	9762	19522	15094	12192	9650	19218	38878	74406	20941	N/A

Table 5.10 – f150s: statistics with half-idling (H) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

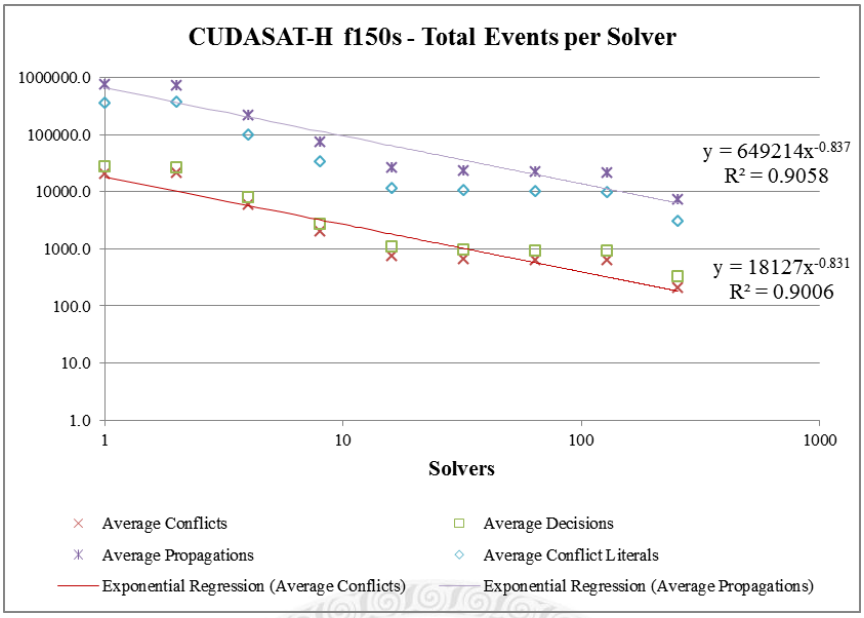


Figure 5.31 - f150s: total events per solver of with half-idling (H) configuration

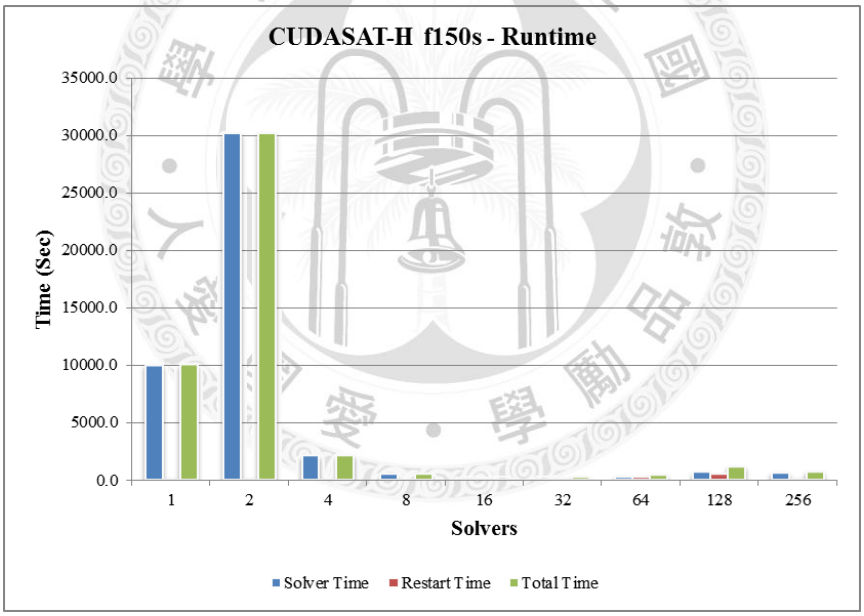


Figure 5.32 - f150s: runtime with half-idling (H) configuration

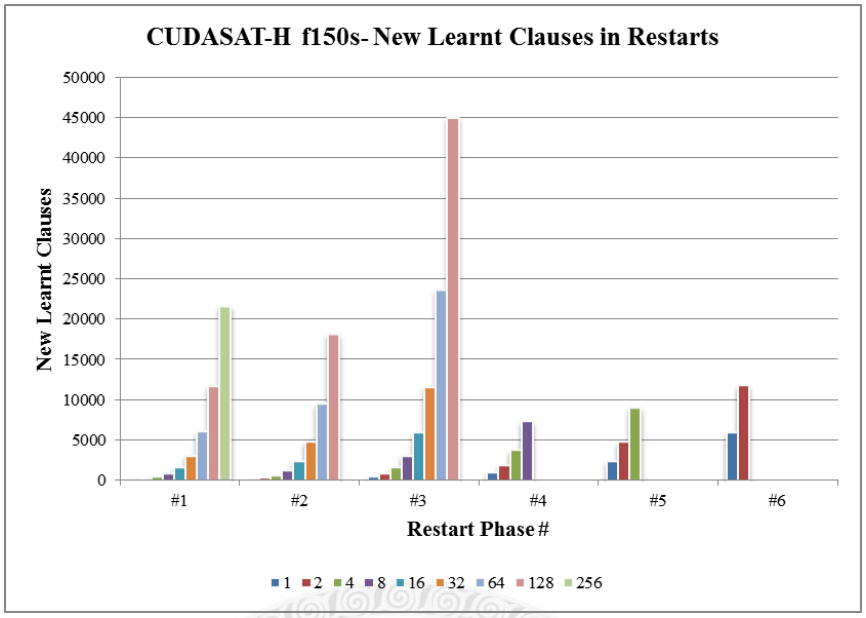


Figure 5.33 - f150s: the new learnt clauses in each restart with half-idling (H) configuration

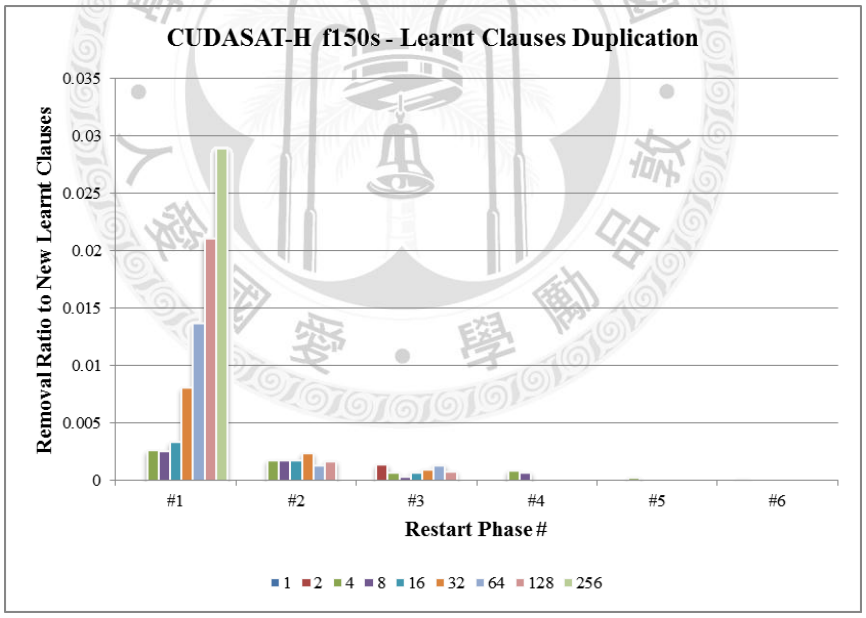


Figure 5.34 - f150s: the duplicated ratio in each restart with half-idling (H) configuration

■ f150s with no-idling (N) configuration.

	CUDASAT-N									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	6	6	5	4	3	3	3	3	3	10
Conflicts	20774	42181	24194	16064	10330	22133	42833	77904	74092	1541
Average Conflicts	20774.0	21090.5	6048.5	2008.0	**645.6**	**691.7**	**669.3**	**608.6**	**289.4**	1541
Conflicts Speed (/sec)	2.1	1.4	11.3	39.3	98.7	147.6	184.8	150.5	60.9	57074
Decisions	27620	53367	32115	21519	15220	31929	62119	115261	109517	1864
Average Decisions	27620.0	26683.5	8028.8	2689.9	**951.3**	**997.8**	**970.6**	**900.5**	**427.8**	1864
Decisions Speed (/sec)	2.8	1.7	15.0	52.6	145.4	213.0	267.9	222.7	90.0	69037
Propagations	742023	1468428	868983	575902	369148	783833	1496097	2685430	2549107	49636
Average Propagations	742023.0	734214.0	217245.8	71987.8	**23071.8**	**24494.8**	**23376.5**	**20979.9**	**9957.4**	49636
Propagations Speed (/sec)	74.5	47.7	406.7	1407.9	3527.1	5227.9	6453.4	5187.6	2095.8	1838370
Conflict Literals	347579	743011	396459	263287	155907	346542	676254	1222836	1163382	13762
Average Conflict Literals	347579.0	371505.5	99114.8	32910.9	**9744.2**	**10829.4**	**10566.5**	**9553.4**	**4544.5**	13762
Solver Time (sec)	9965.6	30755.5	2136.6	409.1	104.7	149.9	231.8	517.7	1216.3	0.027
Restart Time (sec)	8.7	31.7	22.7	16.6	14.3	41.3	129.1	378.5	592.8	N/A
Total Time (sec)	9974.5	30793.6	2161.3	426.5	119.7	193.0	366.5	918.2	1863.5	0.027
Solver Ratio	0.999	0.999	0.989	0.959	0.875	0.777	0.633	0.564	0.653	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1272	1211	1089	845	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	9762	19217	14702	10511	8325	16362	32637	63716	65371	N/A

Table 5.11 - f150s: statistics with no-idling (N) configuration

* The underlined boldface numbers are less than MiniSAT-2.2.0.

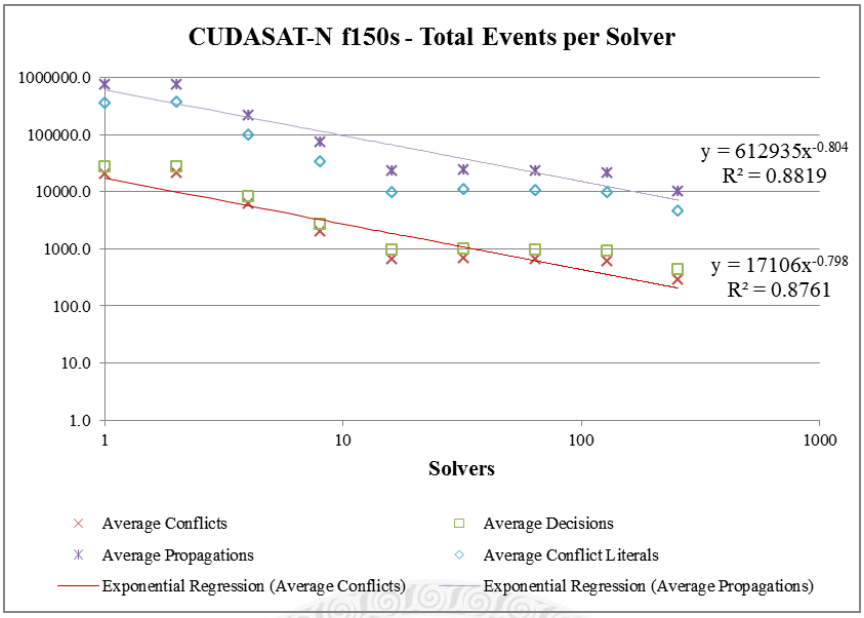


Figure 5.35 - f150s: total events per solver of with no-idling (N) configuration

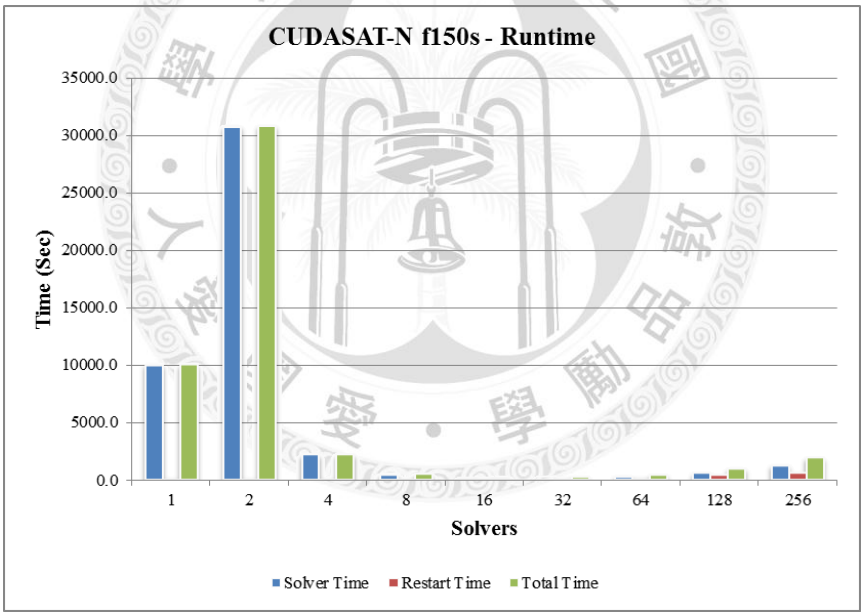


Figure 5.36 - f150s: runtime with no-idling (N) configuration

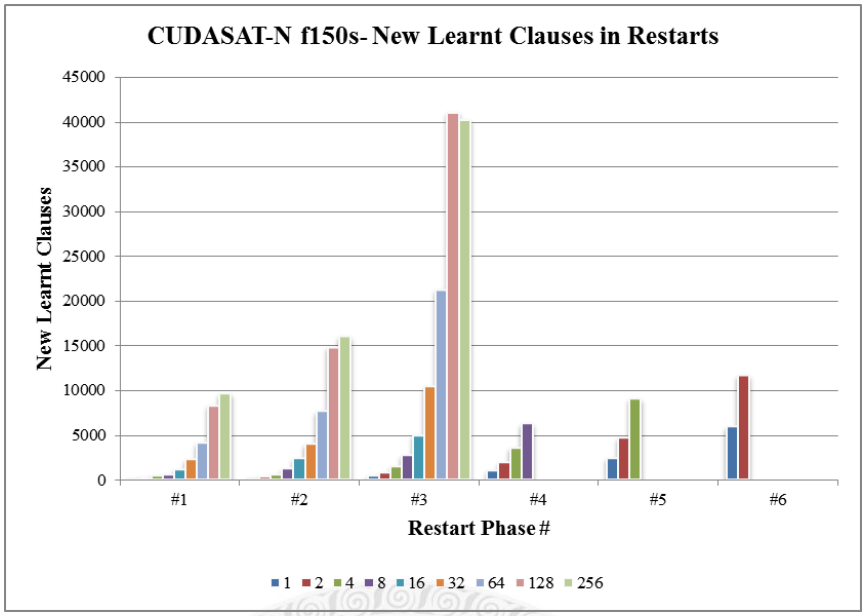


Figure 5.37 - f150s: the new learnt clauses in each restart with no-idling (N) configuration

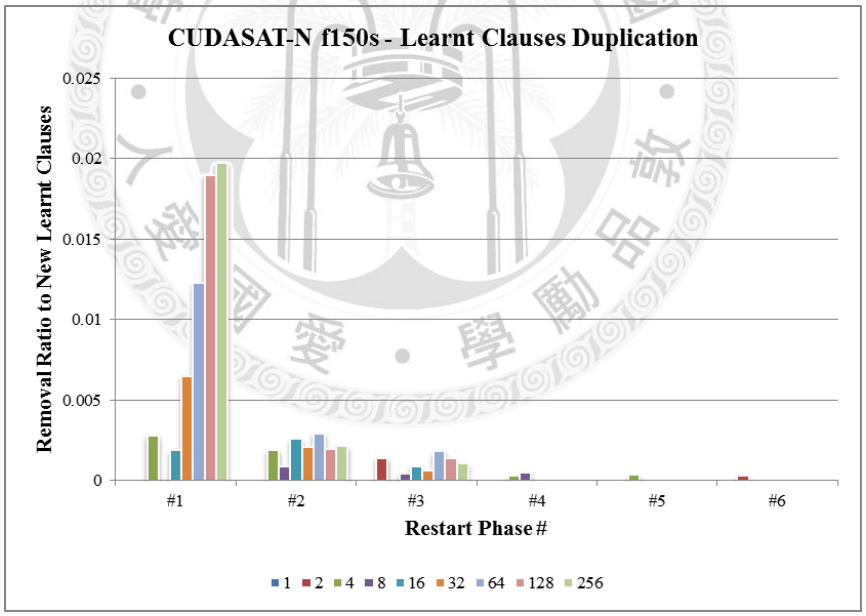


Figure 5.38 - f150s: the duplicated ratio in each restart with no-idling (N) configuration

5.3.8 Comparison of Configurations: f150s

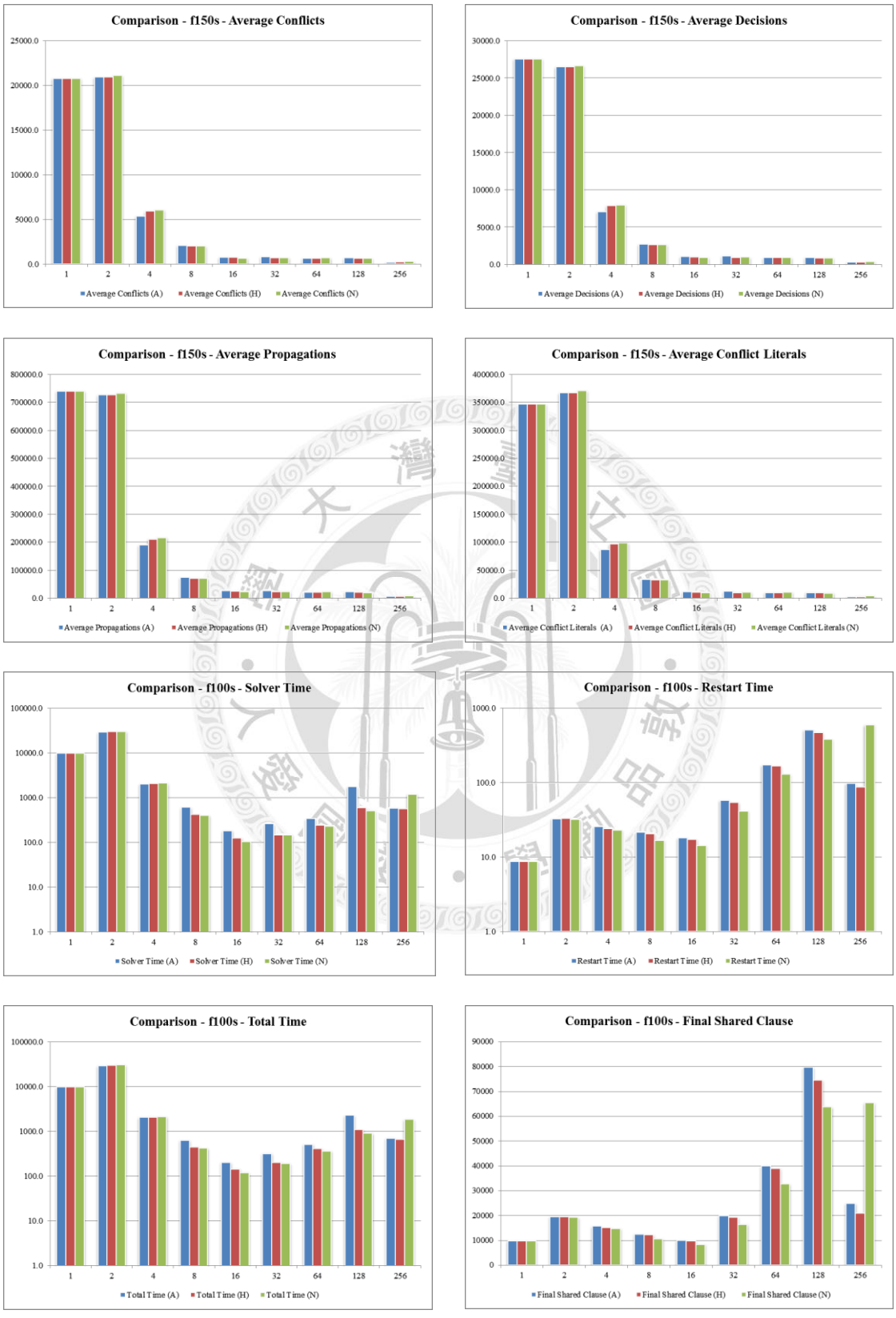


Figure 5.39 – f150s: comparison of configuration A, H, and N

5.3.9 Discussion: f150s

- Configuration A

The single and two solvers have long search processes, which needs 6 restarts before finding the solution. While other solvers need much shorter process. In 256 solvers, there is one lucky solver which gets the solution early in the second solver phase.

The duplication of learnt clauses is below 3% in the first restart phase, which indicates a good divergence of solvers.

- Configuration H

The half-idling (H) configuration saves some runtime except for 256 solvers. Since the lucky solver in 256 solvers enters restart phase earlier than in configuration A, the searching process is lengthened.

- Configuration N

The lucky solver in 256 solvers is not lucky anymore because it enters restart phase much earlier than in configuration H. The search process of that solver is altered by later shared clauses.

- Comparison of Configuration A, H, and N

Since 1, 2 and 256 solvers are extreme cases, we could only compare from 4 to 128 solvers. All configurations are generally comparable in average events per solver and runtime.

5.3.10 Random UNSAT Problem: f100u

- f100u is a random generated hard-to-UNSAT problem. It has 100 variables and 440 clauses. Unsatisfiable.
- f100u with all-idling (A) configuration.

	CUDASAT-A									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	4	4	4	3	3	3	3	2	1	4
Conflicts	3286	5628	8672	11045	15010	20802	40044	53215	63870	423
Average Conflicts	3286.0	2814.0	2168.0	1380.6	938.1	650.1	625.7	**415.7**	**249.5**	423
Conflicts Speed (/sec)	12.4	18.9	37.3	90.6	145.7	153.5	182.9	208.6	118.4	28200
Decisions	4201	6856	10965	13962	19215	27158	52076	70089	87267	493
Average Decisions	4201.0	3428.0	2741.3	1745.3	1200.9	848.7	813.7	547.6	**340.9**	493
Decisions Speed (/sec)	15.9	23.0	47.2	114.5	186.5	200.4	237.9	274.8	161.8	32867
Propagations	90293	152286	234723	299866	393746	543760	1038441	1342135	1633424	10056
Average Propagations	90293.0	76143.0	58680.8	37483.3	24609.1	16992.5	16225.6	10485.4	**6380.6**	10056
Propagations Speed (/sec)	342.1	510.8	1010.7	2459.7	3821.8	4012.3	4744.2	5261.4	3028.7	670400
Conflict Literals	34829	60924	92929	115888	155885	215200	423768	537066	649758	2412
Average Conflict Literals	34829.0	30462.0	23232.3	14486.0	9742.8	6725.0	6621.4	4195.8	2538.1	2412
Solver Time (sec)	263.9	298.1	232.2	121.9	103.0	135.5	218.9	255.1	539.3	0.015
Restart Time (sec)	0.4	1.4	5.9	3.7	11.9	50.2	322.6	139.6	53.2	N/A
Total Time (sec)	264.5	300.0	238.6	125.9	115.6	187.1	544.2	403.3	612.2	0.015
Solver Ratio	0.998	0.994	0.973	0.968	0.891	0.724	0.402	0.633	0.881	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1273	1211	1090	846	N/A
Completed Instances	1	1	1	1	1	1	44	1	1	1
Final Shared Clause	1562	3121	3686	4990	9967	15839	12099	28026	24453	N/A

Table 5.12 - f100u: statistics with all-idling (A) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

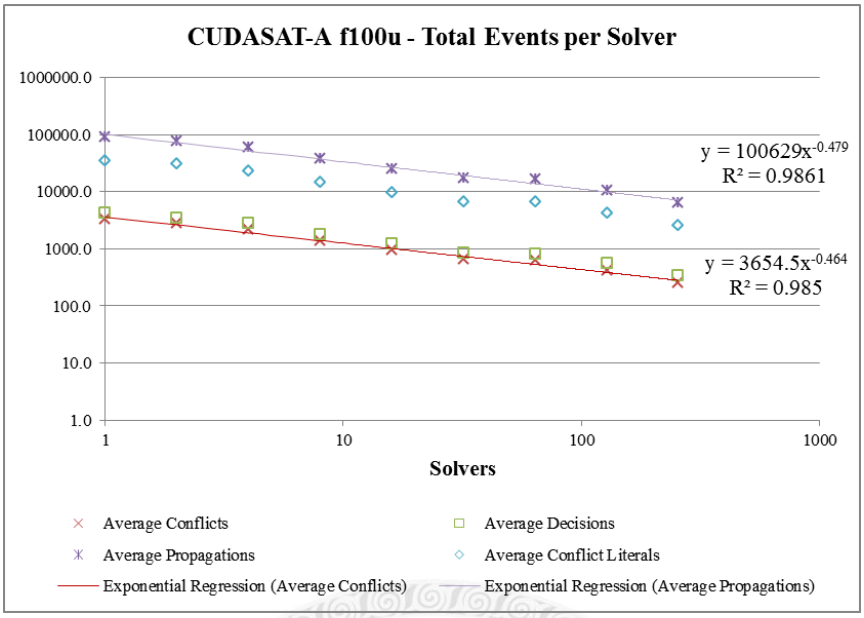


Figure 5.40 - f100u: total events per solver of with all-idling (A) configuration

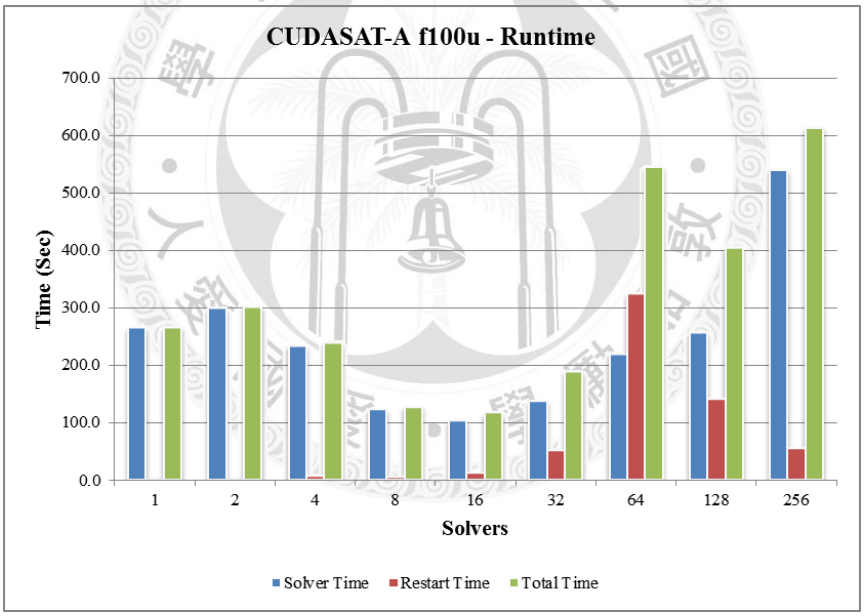


Figure 5.41 - f100u: runtime with all-idling (A) configuration

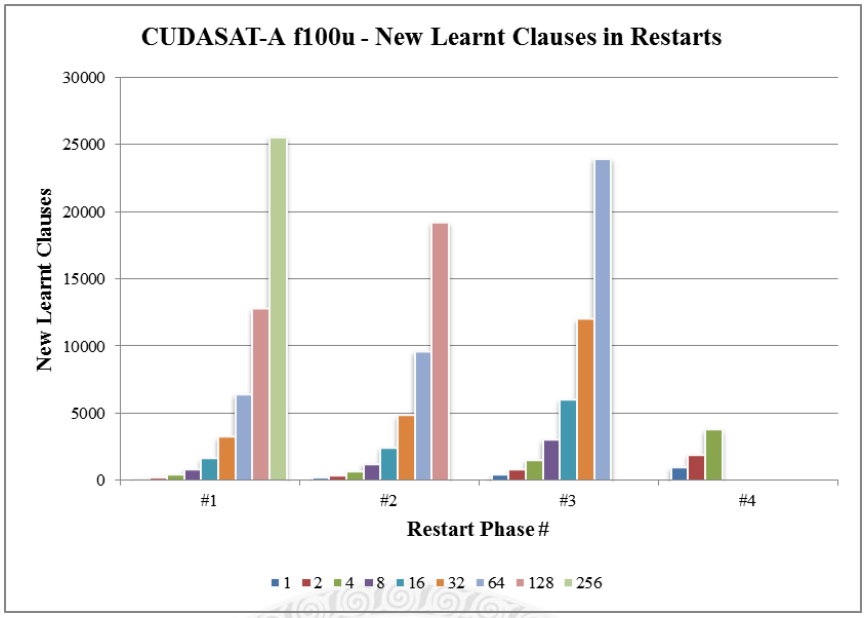


Figure 5.42 - f100u: the new learnt clauses in each restart with all-idling (A) configuration

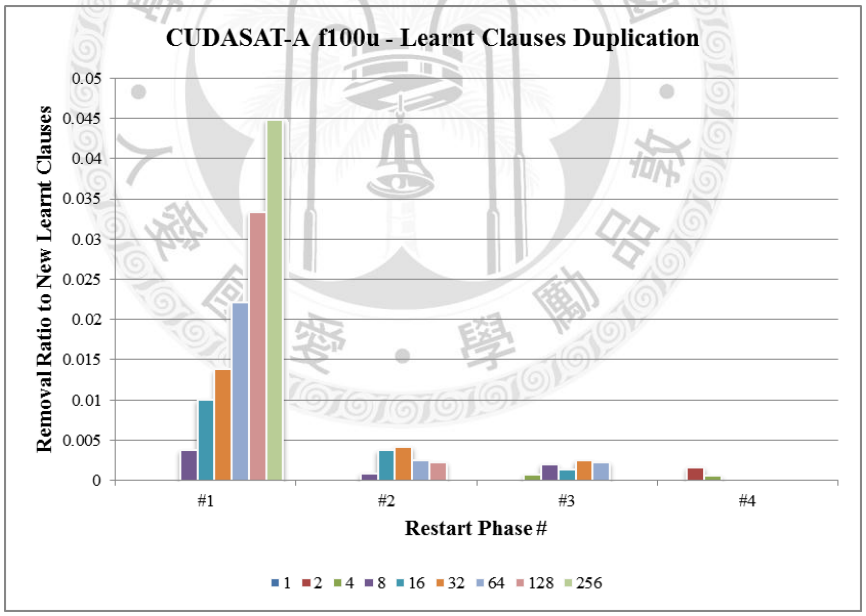


Figure 5.43 - f100u: the duplicated ratio in each restart with all-idling (A) configuration

■ f100u with half-idling (H) configuration.

	CUDASAT-H									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	4	4	4	3	3	3	3	2	2	4
Conflicts	3286	5623	8609	11320	15230	20521	39238	45347	59380	423
Average Conflicts	3286.0	2811.5	2152.3	1415.0	951.9	641.3	613.1	**354.3**	**232.0**	423
Conflicts Speed (/sec)	12.4	18.8	42.1	96.2	192.0	247.7	264.6	359.0	126.4	28200
Decisions	4201	6849	10857	14345	19419	26555	51044	60411	81831	493
Average Decisions	4201.0	3424.5	2714.3	1793.1	1213.7	829.8	797.6	**472.0**	**319.7**	493
Decisions Speed (/sec)	15.9	22.9	53.1	121.9	244.8	320.5	344.2	478.3	174.3	32867
Propagations	90293	152113	232299	305431	398959	534778	1014686	1152236	1510830	10056
Average Propagations	90293.0	76056.5	58074.8	38178.9	24934.9	16711.8	15854.5	**9001.8**	**5901.7**	10056
Propagations Speed (/sec)	342.1	508.9	1136.6	2595.0	5029.7	6455.3	6842.5	9122.3	3217.2	670400
Conflict Literals	34829	60881	91656	118738	158224	213143	415113	460004	604484	2412
Average Conflict Literals	34829.0	30440.5	22914.0	14842.3	9889.0	6660.7	6486.1	3593.8	**2361.3**	2412
Solver Time (sec)	263.9	298.9	204.4	117.7	79.3	82.8	148.3	126.3	469.6	0.015
Restart Time (sec)	0.4	1.4	5.7	3.5	12.8	42.8	257.1	133.1	352.0	N/A
Total Time (sec)	264.5	300.8	210.5	121.5	92.9	127.0	408.8	268.8	846.6	0.015
Solver Ratio	0.998	0.994	0.971	0.968	0.854	0.652	0.363	0.470	0.555	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1273	1211	1090	846	N/A
Completed Instances	1	1	1	1	1	1	1	1	1	1
Final Shared Clause	1562	3121	3650	4923	8701	16981	18757	25964	50850	N/A

Table 5.13 – f100u: statistics with half-idling (H) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

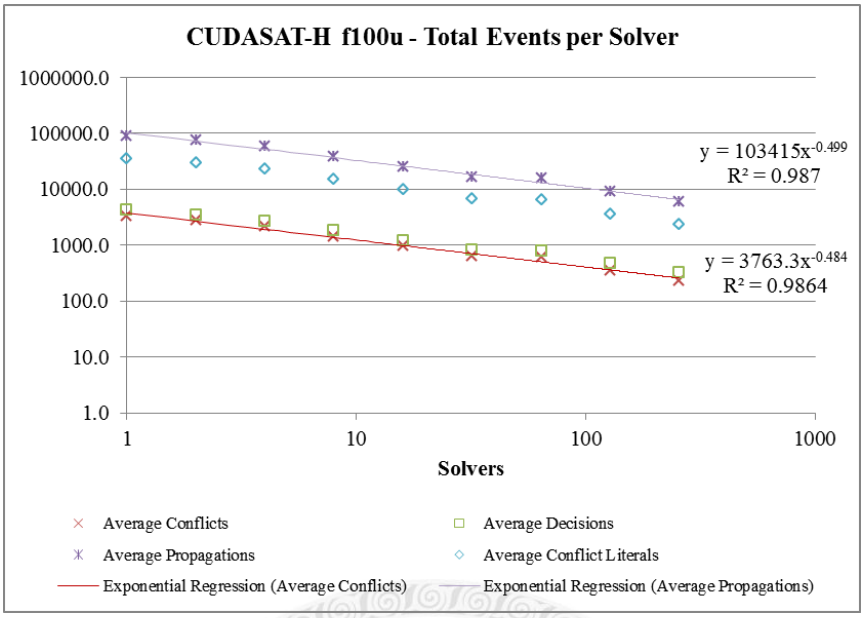


Figure 5.44 - f100u: total events per solver of with half-idling (H) configuration

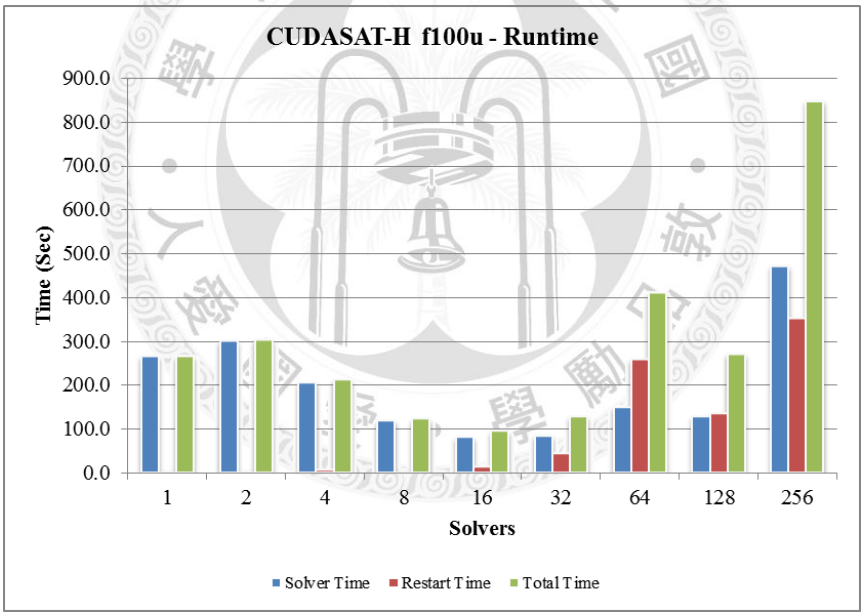


Figure 5.45 - f100u: runtime with half-idling (H) configuration

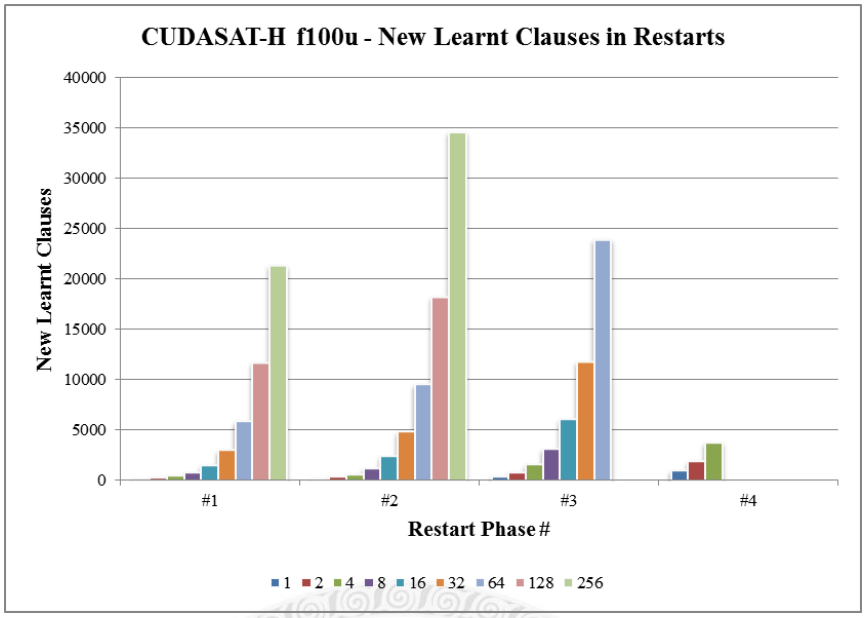


Figure 5.46 - f100u: the new learnt clauses in each restart with half-idling (H) configuration

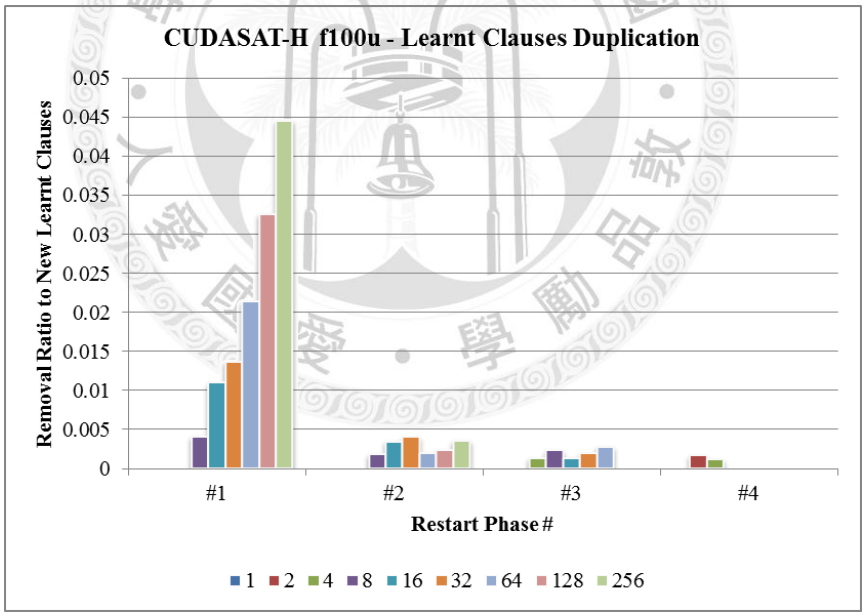


Figure 5.47 - f100u: the duplicated ratio in each restart with half-idling (H) configuration

■ f100u with no-idling (N) configuration.

	CUDASAT-N									MiniSAT -2.2.0
Solvers	1	2	4	8	16	32	64	128	256	1
Restarts	4	4	4	4	3	3	3	2	2	4
Conflicts	3286	5861	8556	11660	15766	20611	34626	57904	53574	423
Average Conflicts	3286.0	2930.5	2139.0	1457.5	985.4	644.1	541.0	452.4	**209.3**	423
Conflicts Speed (/sec)	12.4	19.6	43.2	95.7	201.3	252.4	309.4	299.0	102.9	28200
Decisions	4201	7120	10906	14854	20003	26793	45464	74882	69882	493
Average Decisions	4201.0	3560.0	2726.5	1856.8	1250.2	837.3	710.4	585.0	**273.0**	493
Decisions Speed (/sec)	15.9	23.7	55.1	122.0	255.4	328.1	406.3	386.7	134.3	32867
Propagations	90293	157953	230863	315730	415613	542639	903414	1464377	1358434	10056
Average Propagations	90293.0	78976.5	57715.8	39466.3	25975.8	16957.5	14115.8	11440.4	**5306.4**	10056
Propagations Speed (/sec)	342.1	526.9	1165.3	2592.5	5306.9	6644.6	8073.7	7561.9	2610.0	670400
Conflict Literals	34829	63667	91232	122269	163269	211278	368300	590378	544265	2412
Average Conflict Literals	34829.0	31833.5	22808.0	15283.6	10204.3	6602.4	5754.7	4612.3	**2126.0**	2412
Solver Time (sec)	263.9	299.8	198.1	121.8	78.3	81.7	111.9	193.7	520.5	0.015
Restart Time (sec)	0.4	1.4	5.4	24.2	9.7	31.4	176.6	100.2	174.5	N/A
Total Time (sec)	264.5	301.6	204.0	146.4	88.6	114.4	291.3	301.1	710.8	0.015
Solver Ratio	0.998	0.994	0.971	0.832	0.884	0.714	0.384	0.643	0.732	N/A
Memory (MB)	1402	1402	1402	1402	1403	1405	1408	1414	1427	N/A
Heap Size (MB)	1332	1330	1326	1319	1303	1273	1211	1090	846	N/A
Completed Instances	1	1	1	1	1	1	14	1	1	1
Final Shared Clause	1562	3048	3468	3694	8828	16080	21522	20731	21442	N/A

Table 5.14 - f100u: statistics with no-idling (N) configuration

* The underlined boldface values are less than MiniSAT-2.2.0.

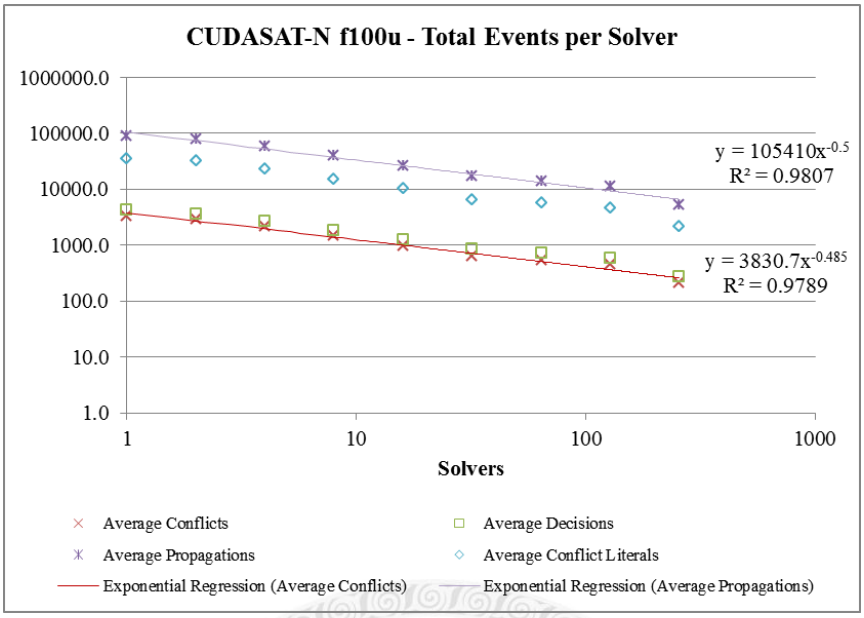


Figure 5.48 - f100u: total events per solver of with no-idling (N) configuration

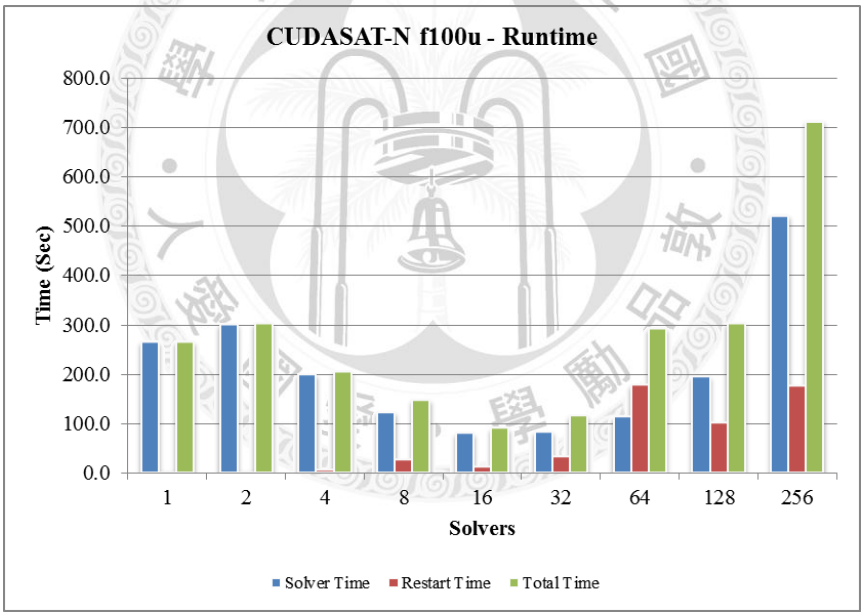


Figure 5.49 - f100u: runtime with no-idling (N) configuration

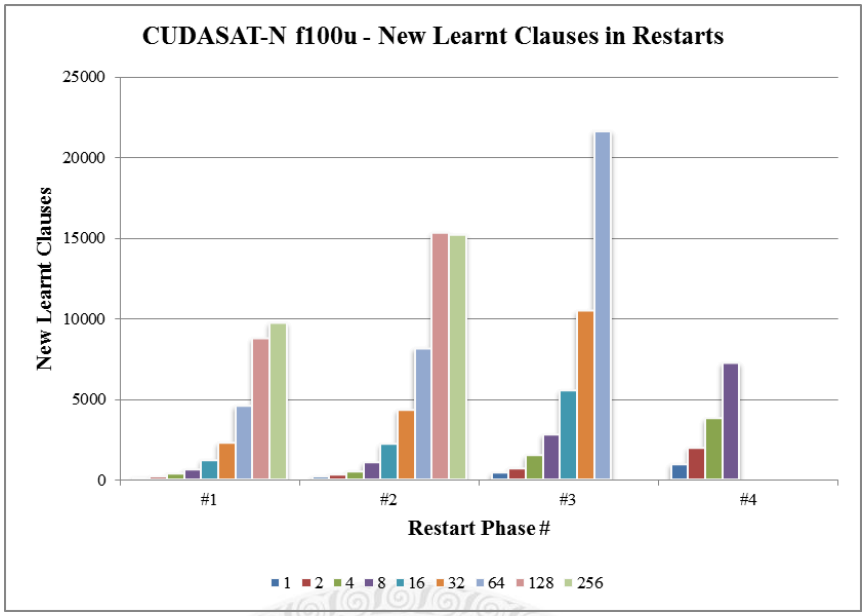


Figure 5.50 - f100u: the new learnt clauses in each restart with no-idling (N) configuration

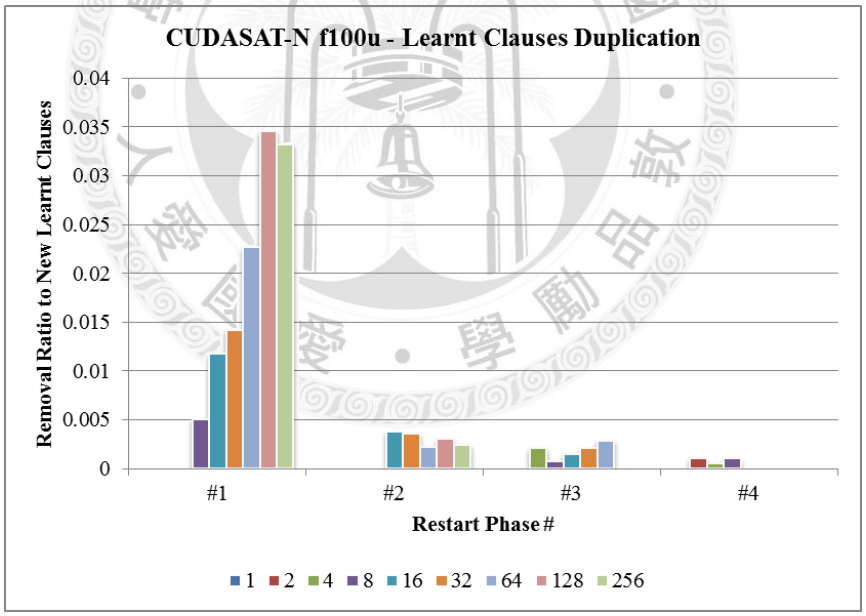


Figure 5.51 - f100u: the duplicated ratio in each restart with no-idling (N) configuration

5.3.11 Comparison of Configurations: f100u

Figure 5.52 - f100u: comparison of configuration A, H, and N

5.3.12 Discussion: f100u

- Configuration A

  In this hardest case, the downward trend in the events per solver seems to be regular.

- Configuration H

  The downward trend is similar to configuration A.

- Configuration N

  In general, configuration N still saves significant runtime in most cases.

- Comparison of Configuration A, H, and N

  Like qg4-08, the average events per solver, runtime, and shared clauses are significantly lower in configuration N in most cases, others are comparable. It maintains a fairly good learnt clause quality here.

Chapter 6 Conclusions and Future Work

We have implemented a massive parallel CDCL-DPLL SAT solver with clause sharing on CUDA. CUDASAT has demonstrated that with sharing clauses, there is a downward trend in average searching events per solver when increasing the number of parallel solvers.

The overhead of CUDASAT lies in thread management, data synchronization, and shared clause processing. It grows rapidly when solver instances increases due to more dynamic device memory de-allocation and allocation, which is extremely slow. Thus the runtime increases significantly beyond 32 solver instances in general.

CUDASAT is not comparable to state-of-the-art parallel SAT solvers. It serves as a prototype to massive parallelization of SAT solving.

The performance of CUDASAT could be further improved by code optimization, pre-processing, in-processing, and other advanced SAT solving techniques. The collaboration of CPU and GPU might be promising, but requires a re-designing of algorithms and program architecture. CUDASAT might provide a source of high quality learnt clauses to CPU SAT solvers by working concurrently as a master-slave mode. Multi-GPU, multi-stream architecture is also a good method for scalability.

Reference

[1] Martin Davis and Hilary Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol.7, no.3, pp.201-215, July 1960.

[2] Martin Davis, Geoge Logemann, and Donald Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol.5, no.7, pp.394-397, July 1962.

[3] João P. Marques Silva and Karem A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, Los Alamitos, California, United States, pp.220-227, 1996.

[4] João P. Marques-Silva and Karem A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol.48, no.5, pp.506-521, May 1999.

[5] Joao Marques-Silva, Ines Lynce, and Sharad Malik, "Conflict-Driven Clause Learning SAT Solvers," *Handbook of Satisfiability*, Armin Biere et al., Eds.: IOS Press, 2009, ch. 4, pp.131-153.

[6] Carla P. Gomes, Bart Selman, and Henry Kautz, "Boosting combinatorial search through randomization," *Proceedings of the 15th national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, Madison, Wisconsin, United States, pp.431-437, 1998.

[7] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, "Chaff: Engineering an Efficient SAT Solver," *Proceedings of Design*

Automation Conference, 2001., Las Vegas, Nevada, United States, pp.530-535, June 2001.

[8] Armin Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," Technical Report 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr, 69, 4040 Linz, Austria, 2010.

[9] (2009, September) SAT 2009 competitive events booklet: preliminary version. [Online]. http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf

[10] Mate Soos. (2010, September) Homepage of Mate Soos. [Online]. http://www.msoos.org/creating-a-sat-solver-from-scratch

[11] NVIDIA Corporation. (2011, May) CUDA C Programming Guide Version 4.0.

[12] Gunnar Stålmarck, "A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula," Swedish Patent No. 467076 (1992), U.S. Patent No. 5276897 (1994), European Patent No. 0403454 (1995), 1992.

[13] Gunnar Stålmarck, A Proof Theoretic Concept of Tautological Hardness, 1994, Unpublished manuscript.

[14] Mary Sheeran and Gunnar Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic," *Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science*, vol.1522, pp.82-99, 1998.

[15] Bart Selman, Hector Levesque, and David Mitchell, "A New Method for Solving Hard Satisfiability Problems," *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, California, United States, pp.440-446, 1992.

[16] Bart Selman, Henry A. Kautz, and Bram Cohen, “Noise Strategies for Improving Local Search,” *Proceedings of the 12th national conference on Artificial intelligence (vol. 1)*, Seattle, Washington, United States, pp.337-343, 1994.

[17] (2009) The International SAT Competition 2009. [Online]. http://www.satcompetition.org/2009/

[18] (2010) SAT-Race 2010. [Online]. http://baldur.iti.uka.de/sat-race-2010/

[19] Youssef Hamadi, Said Jabbour, and Lakhdar Sais, “ManySat: solver description,” Technical Report, Microsoft Research, 2008.

[20] Sripriya G, Alan Bundy, and Alan Smaill, “Concurrent-Distributed Programming Techniques for SAT Using DPLL-Stålmarck,” *International Conference on High Performance Computing & Simulation, 2009*, Leipzig, Germany, pp.168-175, 2009.

[21] Alex S. Fukunaga, “Massively parallel evolution of SAT heuristics,” *IEEE Congress on Evolutionary Computation, 2009*, Trondheim, Norway, pp.1478-1485, 2009.

[22] Kanupriya Gulati, Paul Suganth, Sunil P. Khatri, Srinivas Patil, and Abhijit Jas, “FPGA-based hardware acceleration for Boolean satisfiability,” *Journal of ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol.14, no.2, pp.1084-4309, April 2009.

[23] Kanupriya Gulati and Sunil P. Khatri, “Boolean satisfiability on a graphics processor,” *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, Providence, Rhode Island, USA, pp.123-126, 2010.

[24] A. Braunstein, M. Mézard, and R. Zecchina, “Survey Propagation: An Algorithm

for Satisfiability," *Random Structures & Algorithms*, vol.27, no.2, pp.201-226, March 2005.

[25] Joël Chavas, Cyril Furtlehner, Marc Mézard, and Riccardo Zecchina, "Survey-propagation decimation through distributed local computations," *Journal of Statistical Mechanics: Theory and Experiment*, vol.2005, no.11, November 2005.

[26] Niklas Eén and Niklas Sörensson, "An Extensible SAT-solver," *6th International Conference on Theory and Applications of Satisfiability Testing*, Santa Margherita Ligure, Italy, vol.2919/2004 333-336, pp.502-518, 2003.

[27] Quirin Meyer, Fabian Schönfeld, Marc Stamminger, and Rolf Wanka, "3-SAT on CUDA: Towards a Massively Parallel SAT Solver," *2010 International Conference on High Performance Computing and Simulation (HPCS)*, Caen, France, pp.306-313, 2010.

[28] Hervé Deleau, Christophe Jaillet, and Michaël Krajecki, "GPU4SAT: solving the SAT problem on GPU," *Proceedings of 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, 2008.

[29] Google Incorporation. Google code. [Online]. http://code.google.com/

[30] Stephen A. Cook, "The complexity of theorem-proving procedures," *Proceedings of the third annual ACM symposium on Theory of computing*, Shaker Heights, Ohio, United States, pp.151-158, 1971.

[31] Leonid Anatolievich Levin, "Universal'nyĭe pereborny ĭe zadachi (Universal Sequential Search Problems, in Russian)," *Problemy Peredachi Informatsii*, vol.9, no.3, pp.115-116, 1973.

[32] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä, “Incorporating Clause Learning in Grid-Based Randomized SAT Solving,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol.6, pp.223-244, June 2009.

[33] Kei Ohmura and Kazunori Ueda, “c-sat: A Parallel SAT Solver for Clusters,” *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, Swansea, United Kingdom, pp.524-537, 2009.

[34] Institute of Electrical and Electronics Engineers, Inc. (1996, October) IEEE Standards Interpretations for IEEE Std 1003.1c-1995. [Online]. http://standards.ieee.org/findstds/interps/1003-1c-95_int/index.html

[35] Leonardo Dagum and Ramesh Menon, “OpenMP: An Industry Standard API for Shared-Memory Programming,” *Computational Science & Engineering, IEEE*, vol.5, no.1, pp.46-55, January-March 1998.

[36] NVIDIA Corporation. (2009, October) NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

[37] David H. Wolpert and Kagan Tumer, “An Introduction to Collective Intelligence,” Technical Report, Ames Research Center, Intelligent Systems Division, NASA, Moett Field, 1999.

[38] Ramin Zabih and David McAllester, “A rearrangement search strategy for determining propositional satisfiability,” *National Conference on Artificial Intelligence*, pp.155-160, July 1988.

[39] João P. Marques-Silva and Karem A. Sakallah, “Boolean Satisfiability in Electronic Design Automation,” *Proceedings of the 37th Annual Design Automation Conference*, Los Angeles, California, United States, pp.675-680, 2000.

[40] Paul Beame, Henry Kautz, and Ashish Sabharwal, "Towards Understanding and Harnessing the Potential of Clause Learning," *Journal of Artificial Intelligence Research*, vol.22, no.1, pp.319-351, July 2004.

[41] Robert Tarjan, "Finding Dominators in Directed Graphs," *Society for Industrial and Applied Mathematics Journal on Computing*, vol.3, no.1, pp.62-89, March 1974.

[42] João P. Marques Silva and Karem A. Sakallah, "Dynamic search-space pruning techniques in path sensitization," *Proceedings of the 31st annual Design Automation Conference*, San Diego, California, United States, pp.705-711, 1994.

[43] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik, "Efficient conflict driven learning in a boolean satisfiability solver," *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, San Jose, California, United States, pp.279-285, 2001.

[44] Sharad Malik. (2004) Sharad Malik. [Online]. http://www.princeton.edu/~sharad/CMUSATSeminar.pdf

[45] Holger H. Hoos and Thomas Stützle, "SATLIB: An Online Resource for Research on SAT," *Sat2000: highlights of satisfiability research in the year 2000*, Ian Gent, Hans van Maaren, and Toby Walsh, Eds. Amsterdam, The Netherlands: IOS Press, 2000, pp.283-292, SATLIB is available online at www.satlib.org.

[46] (2000, August) SATLIB - Benchmark Problems. [Online]. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html