Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Máster en Métodos Formales en Ingeniería Informática

Trabajo Fin de Máster

SAT Solving via Algebraic Normal Form and GPU

Autor: Ignacio Ballesteros González
Tutor: Julio Mariño Carballo

Madrid, 16 de Julio de 2021

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Métodos Formales en Ingeniería Informática

SAT Solving via Algebraic Normal Form and GPU

14 de Julio de 2021

Autor: Ignacio Ballesteros González
Tutor: Julio Mariño Carballo
ETSI Informáticos
Universidad Politécnica de Madrid

I can't find an efficient algorithm, but neither can all these famous people.

Computers and Intractability; A Guide to the Theory of NP-Completeness
Michael R. Garey and David S. Johnson

Contents

Abstract

Boolean Satisfiability Problem (SAT) is the first problem in Computer Science to have been proven NP-Complete. We can encode complex problems as SAT, and its applications are spread across many fields. The application of SAT reaches anything that we can express as a problem of constraints, from circuit verification to bounded model checking. Any efficient algorithm applied to solving SAT can have a deep impact in all of its applications, so SAT solvers have been steadily improving, supporting complex models that were unimaginable to solve a couple of decades ago.

These SAT solvers have been mainly based on backtracking algorithms, including conflict clause analysis to efficiently traverse the search space. The implementation of efficient data structures and clever heuristics made possible the development of competitive SAT solvers.

Computer architecture has played a major role in the evolution of SAT solvers. The specialization of parallel and distributed algorithms for SAT solving has open new competition branches for solvers. Hardware-accelerated solvers open new opportunities for efficient implementations, but the challenges that arise in these architectures, like GPUs, are difficult to fit in the traditional algorithms.

In this work we study a case of SAT solving in GPU. SAT solvers commonly digest a boolean formula in Conjunctive normal form (CNF). This encoding has great properties when used in backtracking solvers, which are commonly designed as sequential algorithms. However, there are other boolean formula representations with better properties for parallel algorithms. Algebraic normal form (ANF) offers an exploitable parallel operation that can be efficiently done in GPU. ANF, also known as Zhegalkin polynomial, is an algebra representing boolean formulas as a polynomial over $\mathbb{Z}_2$.

We have developed a tool that pre-processes CNF and propositional formulas, converting them to a set of ANF constraints. We check satisfiability by finding the 0 polynomial, the only ANF representation that is unsatisfiable. This process is done as an iteration of two steps: 1.- Multiply two polynomials and, 2.- Reduce the resulting polynomial to canonical ANF. These polynomials tends to grow exponentially in size as we multiply them with new constraints. We have developed parallel algorithms that take advantages of the GPU architecture and speed up both stages. This study improves the development of a previous solver based on this technique, overcoming challenges in the implementation of these methods and with promising parallelization results. Further development could be done to apply this tool as a component of existing SAT solvers.

Resumen

El problema de satisfacibilidad booleana (SAT) fue el primer problema en demostrarse NP-Completo. Muchos problemas complejos se resuelven reduciéndolos primero a problemas SAT, así que sus aplicaciones pueden alcanzar varias disciplinas. Todo problema que se pueda expresar mediante restricciones puede ser resuelto por SAT, como por ejemplo la verificación de circuitos o de software. Cualquier algoritmo aplicado a SAT que resulte ser eficiente tiene un profundo potencial de impacto en cualquiera de sus aplicaciones. Es por esto que los SAT *solvers* se han ido mejorando de manera constante, con la capacidad de abordar modelos cada vez más complejos inimaginables de resolver hace solo unas décadas.

Los SAT *solvers* se basan generalmente en algoritmos de *backtracking* a los que se les ha añadido análisis de conflictos para recorrer de una forma más eficiente el espacio de búsqueda. Todo esto sumado al desarrollo de estructuras de datos eficientes y al diseño de heurísticas avanzadas han logrado posicionar a los SAT *solvers* como herramientas muy competitivas.

El avance en nuevas arquitecturas también ha tenido un rol importante en la evolución de los SAT *solvers*. La especialización de los algoritmos paralelos y distribuidos para SAT ha abierto nuevas ramas donde estos *solvers* pueden competir. Aquellos que se han basado en la aceleración por *hardware* han creado oportunidades para lograr algoritmos más potente y eficientes, pero de la mano han traído los desafíos de adaptar unos algoritmos que no necesariamente encajan en estos componentes como las GPU. En este trabajo presentamos el estudio de un algoritmo para resolver SAT en un entorno altamente paralelo como la GPU.

Los SAT *solvers* suelen partir de una representación de las fórmulas booleanas en forma normal conjuntiva (CNF). Esta representación tiene unas propiedades que se aprovechan satisfactoriamente en los algoritmos de *backtracking*, habitualmente diseñados para la ejecución secuencial. Sin embargo, existen otras representaciones de las fórmulas booleanas con propiedades que pueden encajar mejor en un contexto paralelo. La forma normal algebraica (ANF) es una de estas representaciones que posee una operación altamente paralelizable que podría realizarse en una GPU. Las formulas ANF, también conocidas como polinomios de Zhegalkin, es una representación algebraica booleana en forma de polinomio de números enteros modulo dos ($\mathbb{Z}_2$).

En este trabajo hemos desarrollado una herramienta que aplica un preprocesado a formulas en CNF y proposicionales para convertirlas en un conjunto de restricciones en ANF. Mediante ellas, podemos comprobar las satisfacibilidad de un problema encontrando el polinomio 0, el único que no es satisfacible con la representación ANF. El proceso se realiza mediante dos pasos: 1.- Multiplicación de dos polinomios y 2.- La reducción del polinomio resultante a una forma ANF canónica. Estos polinomios tienden a crecer exponencialmente cuando se les aplican las nuevas restricciones. Para acelerar la ejecución de estos dos pasos hemos desarrollado un algoritmo paralelo que aprovecha los beneficios de componentes como la GPU. Este estudio mejora el desarrollo de un *solver* previo basado en esta misma técnica, superando algunos desafíos que se presentaron en ese caso y con resultado prometedores en su potencial de paralelización. Como desarrollo futuro, se podría tratar de integrar esta herramienta en alguno de los SAT *solvers* ya existentes.

Chapter 1

Introduction

Boolean Satisfaction Problem (SAT) is one of the fundamental problems in Computer Science. It has been widely studied because of its expressiveness, and its applications are widespread across different disciplines. Its versatility lies on the capacity to encode any problem, so find an efficient SAT solver and you will also find an efficient one for many other questions.

In this work, we develop a tool that brings the parallel power of GPU computing to SAT solving using Algebraic Normal Form. We want to explore the benefits and drawbacks of a different architecture that is not commonly used in the context of SAT solving. ANF is the notation providing an exploitable parallel operation.

1.1. SAT Solving

The problem of satisfiability states that given a boolean formula, we want to find a combination of values for its variables that make the formula true, when such a combination exists we say that the formula is satisfiable. The formula is unsatisfiable when no combination can be found.

Boolean satisfaction problems didn't get much attention until Stephen A. Cook proved in 1971 how these kinds of problems could encode any other NP-Complete problem. Previously developed algorithms like the ones of Martin Davis and Hilary Putnam were recovered and improved [17]. The race to achieve an efficient solver was served.

But why is SAT solving important? As we have said, many problems can be solved via SAT. They are easy to encode as Constraint Satisfaction Problems and let a SAT solver does the work. Graph Coloring is a classic problem to be solved via SAT, but there are plenty more examples like circuit verification, factoring integers or bounded model checking [17].

Another frequent scenario that we can easily solve with SAT is version dependency resolution on package managers, an operation that many developers perform several times a day.

Binary Decision Diagrams (BDD) are a great way of representing boolean formulas. This data structure of a formula is a direct acyclic graph between boolean variables, the nodes, and the values 1 and 0 as leaves. These representations are useful for circuit design, testing and verification [3].

The first algorithms developed to find a solution were the ones of Martin Davis and Hilary Putnam, with some adjustments to form the family of DPLL algorithms. These algorithms use backtracking as a way to traverse the tree of possible values assignments. It is not based on BDD representation, instead, it uses a Conjunctive Normal Form (CNF) encoding. CNF is a formula that meets the following requirement: they are conjunction ($\wedge$) of disjunctive clauses ($\vee$). This representation is powerful enough to represent any boolean formula, and we can also convert between other encodings easily. There are more encodings for boolean formulas, with different properties that may suit better our needs. For example, Disjunctive Normal Form (DNF) offers an easy method to check satisfiability but can make operations like negation or conjunction over them more complex. A similar encoding to DNF is Algebraic Normal Form (ANF), which is based on exclusive disjunctions ($\oplus$) of conjunctions. This is the encoding that we choose to work with and We will deeply discuss its properties in Section 2.1.

The algorithm of DPLL uses the CNF encoding to work. This was the first solver that could efficiently find a solution, but more importantly, keep a low memory usage. DPLL algorithm reduces the search space by assigning values to boolean variables. This variable is now removed from the clauses and checked again. In case that we end with an empty formula, a contradiction, we need to backtrack and try with another boolean value. When no more variables are left to assign, we have reached a solution, therefore the formula is satisfiable. When we cannot backtrack more, there is no combination of values and the formula is unsatisfiable. In order to reduce the search space and not trying any possible combination, there are some optimizations applied. The *unit propagation*, when having single variable clauses we know that the value must be *true*. Another rule applied is the *pure literal elimination*, when an atomic formula (with just one variable) is assigned with a single literal value, we can eliminate all other formulas containing the same atomic formula. [9]

1.1.1. Conflict Driven Clause Learning solvers

New solvers evolved from DPLL, trying to be more efficient on backtracking. When we assign a value to a variable that ends up creating an empty formula, a conflict has appeared. The family of Conflict Driven Clause Learning (CDCL) solvers work on these scenarios and try to find what assignment caused the conflict [3] and learn from it. With this information, they are able to backtrack, or "back-jump", in a non-chronological order. Clause learning also

prevents from repeating paths leading to conflicts. Solvers like GRASP [25] and Chaff [22] used these techniques, improving a lot the State of the Art in SAT solving. Chaff focused on optimizing unit propagation, or Boolean Constraint Propagation (BCP), by watching specific formulas that could become unit clauses. They also introduced a new approach to clause deletion, to prevent extended memory usage and restart techniques.

With these new efficient solvers, new SAT competitions were established. During SAT 2002, a track for algorithms for CDCL like Chaff was prepared. They wanted to compare with a set of benchmarks how State of the Art solvers were performing and improving year by year. With DIMACS standard notation, a common set of benchmarks were prepared [16]. DIMACS is the standard notation for CNF clauses used by SAT solvers. The participation in these competitions raised year after year, with different approaches for solvers. They were mainly variations of CDCL solvers and alternative ones, but new parallel ones and portfolios approaches were gaining popularity every year.

With the improvements of solvers and their specialization, new tracks were prepared in these contests to award the best SAT solvers in each category. There are categories as hacks of variations of successful solvers, like MiniSAT for its easiness of tweak [10]. The rise of new hardware architectures had led to the creation of tracks like the parallel one and in recent years, the cloud track for distributed oriented solvers.

Solvers improved the CDCL algorithms by including heuristics on clause learning and elimination, like Glucose [2], a solver based on MiniSAT. Glucose focused on clause learning instead of BCP. They based their approach on giving scores to the discovered conflict clauses. This score was based on the Literal Block Distance (LBD) rule, which rated the clauses by their proximity and dependency. LBD clauses of score 2, of high dependency, were named "glue clauses", from which the solver takes the name. In addition to this heuristic, an aggressive approach to clause elimination was taken. Storing many clauses on memory can detriment the performance of the solver. With the score and measures about used learnt formulas, Glucose can keep the number of clauses low without sacrificing many useful ones.

Current state of the art CDCL solvers have focused again on simplifying data structures and provide a base layer for tweaks and improvements to be applied: CaDiCAL, KISSAT [5]. The family of MapleSAT [20] solvers introduced new heuristics based on machine learning clause analysis. As a comparison of the evolution in SAT solvers, not only the improvements in hardware have helped to solve problem instances faster. The advances in clever algorithms had the same major role in the better performance of these solvers [12].

1.1.2. Parallel SAT solvers

Multiprocessors opened the gate to achieve better performance in new computer architectures. To exploit their power, new algorithms need to be devised. The main problem of sequential CDCL algorithms is the high data dependency between clauses. One way to

avoid data dependency is dividing the search space and lunch the solvers starting from exclusive branches. The alternative approach is sharing the knowledge of clauses between the processors. This is the port-folio methodology. An approach that combines multiple solvers with their own heuristics and restart policies, but at the same time, share some learnt clauses. ManySAT [13] implemented this approach, sharing clauses after periodic restarts of the solvers.

The main approach adopted by SAT solvers where adjusting parts of their algorithms to a parallel approach. MiniSAT have the variation of pMiniSAT [7], keeping the essence of the original MiniSAT: having a simple implementation of already known techniques to serve as a base layer for future tweaks. Glucose, in their version 4 [1], released their algorithm to work on parallel contexts. PLingeling [4] is based on CDCL solvers like Glucose, sharing a common stack of clauses from which processes pool and resolve them.

Last approaches to parallel SAT solving involves, besides variations of sequential solvers, the configuration of solvers. Painless [19] takes this approach of building a framework for parallel SAT solving. They aim to solve the problem of having to implement from scratch a parallel solver, which requires deeper knowledge of concurrency and synchronization mechanisms. A bad implementation could ruin a great parallelization idea. Also, providing a basic structure, it allows building and tuning specific parts of the solver. This strategy has been proven successful with solvers like MiniSAT [10] and CaDiCAL [5]. The final aim of this framework is to provide a way to combine multiple sharing and parallel strategies. This solver has been awarded in the SAT 2016 Competition, winning also in 2017, 2018 and 2020 with derivations of Painless and MapleCOMPSPS [18].

Hardware accelerated SAT solvers

New architectures on computer components can play another role in developing new algorithms for SAT solving. We could consider three categories of hardware accelerated components to analyze: ASICs, FPGAs and GPUs.

Components on DPLL and CDCL algorithms can be efficiently ported to ASICs and FPGAs [26], but they soon reach the limit in the speedup.

1.2. Related work

We have not found literature about a solver that mixes ANF encoding and a parallel algorithm executed on GPU, but there are some works done each of these fields.

1.2.1. GPU Solvers

GPU SAT solvers started to evolve around DPLL approaches, first with a proof of concept in the work of Meyer et al. [21] in 2010 and then the successful approach of Dal Palù in CUD@SAT [24] in 2014. In the same year another tries weren't that successful [8], with implementations that didn't meet the objectives expected in performance, but showed how GPU Sat solving approaches could shift towards CPU-GPU cooperation.

In the end, these approaches seemed that didn't fit well in the domain of GPUs [26], but continuous work kept pushing this parallel approach. GPU implementations were oriented to incomplete SAT algorithms that could help on local search or auxiliary tasks but not with the whole solver. With a promising background of GPU parallelization in typical SAT solving applications (model checking, minimisation or SAT heuristic solving), Osama, Muhammad & Wijs, Anton [23] proposed a successful algorithm for SAT formula simplification in GPUs. There are still challenges to be addressed in the development of parallel SAT solvers [14].

1.2.2. Algebraic normal form solvers

ANF is a boolean formula representation that can be solved as a SAT problem with Gröbner basis [15] or by Gaussian elimination [11]. However, these approaches can't compete with state of the art CDCL solvers.

Bosphorus [6] successfully implemented a clause simplification technique based on conversions between CNF and ANF. This could be applied as a preprocessor for other SAT instances but also as a way to improve clause learning. The techniques for solving includes ANF propagation, similar to BCP, and reductions via Gauss-Jordan Elimination (GJE).

In 2018 David Lilue started developing a parallel SAT solver on GPU using the Zhegalkin polynomial, under the same direction as this Master's Thesis. That work implemented a parallel routine for polynomial multiplication with its own encoding of ANF formulas. As future work, it stated that further development was needed, revisiting the algorithms used and reducing dependency between CPU and GPU.

1.3. Proposal

In this work, we present the study of a partial SAT solver and the resulting tool that preprocesses CNF and propositional formulas, converting them to a set of ANF constraints and performing a satisfiability checking. We execute this check by an incremental accumulation of constraints, done with polynomial multiplication and reduction in a GPU. Our goal is providing a tool for constraint programming, having a satisfiable formula and checking if a new constraint turns it unsatisfiable. In the case of satisfiable instances, our solver leaves a reduced ANF that can be solved via classical methods like Gröbner basis or GJE.

The main challenges [14] that we studied was devising a new encoding of boolean formulas and exploit it in a parallel context. Escaping from already well-studied parallelization algorithms based on CDCL, we tried to see if new paths can be opened to completely and efficiently solve SAT problems with GPUs and ANF. Other scenarios considered could be coupling it with existent solvers and delegating sub-problem solutions.

Structure of the work

In Chapter 2 we define all the terms, concepts and theorems that we are going to reference later. Chapter 3 goes into the details of the implementation, visiting the steps and the process of building the final algorithms. For the outcomes of this study, experimental results are compiled in Chapter 4. Finally, a small review of these results and some future work considerations are included in Chapter 5.

Chapter 2

Preliminaries

2.1. Algebraic normal form

Algebraic Normal Form (ANF) is a boolean formula where the operations between values and variables are only ANDs and XORs. This kind of formula receives multiple names, Zhegalkin Polynomial or Reed-Muller expansion, as the properties of ANF can be studied from different fields.

2.1.1. Properties

In terms of operations, figure 2.1 describes the truth table of ANDs and XORs.

There are some remarkable properties of these operations, equation 2.1 represents 1 as the identity element of AND. Two equivalent terms can be simplified if they are XORed (eq. 2.2). As ANF is equivalente to working in modulo 2 arithmetic, ANDs can be seen as multiplications (eq. 2.3) and XORs as sums (eq. 2.4).

1	A	B	$1 \cdot A$	$1 \oplus A$	$A \cdot B$	$A \oplus B$
1	0	0	0	1	0	0
1	0	1	0	1	0	1
1	1	0	1	0	0	1
1	1	1	1	0	1	0

Figure 2.1: ANDs and XORs truth table

Identity element	$1 \cdot x = x$	(2.1)
$(\oplus)$ Simplification	$x \oplus y \oplus x = y$	(2.2)
Distributive AND	$x(y \oplus z) = xy \oplus xz$	(2.3)
Commutative XOR	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	(2.4)

ANF equivalences

There is a straightforward way to convert from any propositional logic formula to an equivalent ANF.

$$True = 1 \tag{2.5}$$
$$x = x \tag{2.6}$$
$$\neg x = 1 \oplus x \tag{2.7}$$
$$x \wedge y = xy \tag{2.8}$$
$$x \vee y = xy \oplus x \oplus y \tag{2.9}$$
$$x \oplus y = x \oplus y \tag{2.10}$$
$$x \rightarrow y = xy \oplus x \oplus 1 \tag{2.11}$$
$$x \leftrightarrow y = x \oplus y \oplus 1 \tag{2.12}$$

2.1.2. Canonical ANF

A canonical ANF formula is an ANF formula with some properties:

- It is expressed as a XOR of AND formulas.
- Does not contain repeated formulas, as they are canceled by the XOR.

Canonical ANF	Non-canonical ANF
$1 \oplus a \oplus bc \oplus ac$	$1 \oplus a \oplus c(a \oplus b) \oplus a$

Every ANF formula can be reduced to canonical form, which is unique.

Satisfiability. An ANF formula is satisfiable if and only if its canonical form is not empty. As reducing to canonical form is equivalent to finding satisfiability of an ANF formula, we have not reduced the complexity of the problem, but simply change it to a way that hardware acceleration may improve compute times.

2.1.3. ANF encoding

ANF is encoded into representations oriented to parallel processing. In our representation, each monomial will have the same size in bits. Each bit will encode the presence of a variable in the monomial. There is a translation table mapping every variable with their corresponding bit index.

Variable	Index	(8 bit word)
$v1_1$	0	`00000001`
$v1_2$	1	`00000010`
...	...	...

An expression like $v1_1 \wedge v1_2$ will be encoded as `00000011`. The number of variables will determine word's length.

2.1.3.1. Operators in terms of binary encoding

ANF	$Binary$
1	`00`
A	`01`
B	`10`
$1 \cdot A$	`01`
$A \cdot B$	`11`

We cannot perform $1 \oplus x$ in binary encoding.

Sorting. One of the properties of this encoding is that we now have a clear sorting method for ANF that brings us our canonical representation with an strict total in the monomials.

2.1.4. Format Standards

Compatibility is essential to work with logic formula representations. SAT solvers can read different input formats, and we are to describe two of the most extended ones.

2.1.4.1. DIMACS

DIMACS[1] format is a boolean formula representation generally in Conjunctive Normal Form (CNF), a conjunctions of disjunctions. This format is widely used, being the principal format of The International SAT Competition.

This format starts with a series of comments describing the number of variables and clauses that are in the file. Then, it follows a list of clauses in the form of one line per clause, and

[1] `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`

```
p cnf 3 2
1 2 0
1 -3 0
```

Figure 2.2: DIMACS example

each line is a list of positive numbers representing a variable. Each line ends with a 0, and if the number is negative, it means that the variable is negated. Listing 2.2 is an example of a DIMACS file of the formula $(a \vee b) \wedge (a \vee (\neg c))$.

2.1.4.2. SMT-LIB

Widening the scope of SAT solvers, we can get into the lands of SMT. The format for these solvers is based on a Lisp-like syntax, but the standard goes far beyond what DIMACS can do. SMT-LIB has received multiple revisions, including some extensions to work with more powerful theories.

Here we are only going to use the basics of this language as a common interface to multiple SAT problems. Listing 2.3 has the same formula written in DIMACS.

```
(assert (or v1 v2))
(assert (or v1 (not v3)))
```

Figure 2.3: SMT-LIB example

2.2. GPU

Our parallel execution context will be a GPU, where we are going to be performing our multiplication and reduction steps. We need a platform to implement our algorithms and communicate with the GPU. There are several specifications that may be compatible with different GPU models.

We chose the OpenCL platform[2] because it is an open standard, compatible with nearly all GPUs model, while the CUDA toolkit[3] can only be execute on Nvidia models.

This section is intended as a reference guide for the specific concepts that are going to be used during the implementation's explanation.

[2] https://www.khronos.org/opencl/
[3] https://docs.nvidia.com/cuda/

2.2.1. OpenCL platform

OpenCL is an open standard for parallel programming. It provides a common language to different platforms and implementations, leaving to vendors the capability to design compatible devices. From the initial release in 2011 to the third major version in 2020, OpenCL has proven to be a solid platform to a wide range of applications and devices.

Fist implementations of OpenCL aimed at CPUs and GPUs as main platforms, but over the years new devices implemented compatibility with different versions of the standard. Nowadays, OpenCL is supported on different GPU and CPU vendors, mobile platforms and even FPGA.

After years of development, OpenCL has been mainly focused on scientific parallel computing, but the standard has evolved to be compatible with similar frameworks through an intermediate language. The Khronos group have developed an ecosystem of acceleration languages providing compatibility between different applications, from low level heterogeneous computing to graphic rendering and high level applications.

OpenCL is a great choice to implement parallel acceleration as it is a solid platform, compatible with many different devices and great support across libraries and languages.

Terms and concepts

Context. It is the environment in which we are going to execute our OpenCL code. A context is related to a set of devices and objects to interact with the system.

Device. A physical device (CPU, GPU, Mobile, FPGA...) that is able to run under an OpenCL platform implementation. Each version of OpenCL is compatible with previous devices, giving up some performance if the device is not fully compatible. For example, a program written in OpenCL 2.0 that is running on a device designed to support up to version 1.2 is not going to be as optimized as in a proper compatible device.

Platform. OpenCL programs are dependent to each platform implementation. Each platform has specific characteristics: multiple number of cores, memory levels, vector instruction support, and many extensions to improve performance. At runtime, source code must be compiled to the concrete platform, which is an specific version of the supported language version and a set of extensions available.

Programs and kernels. OpenCL code may be loaded from text files or already compiled source code. A program is a collection of functions, and this functions can also be named kernels. Kernels can be enqueued to be executed from the host, from which they can call other functions written in the program.

Queues and events. OpenCL is an event based runtime model, it may even support out-of-order execution. A kernel is commanded to be executed through a queue, and an event handler is returned. We can enqueue more commands that are dependent to the previous kernel execution, and synchronously wait to the desired event to be finished. Multiple queue can co-exist at the same time, operating on different devices and contexts.

Memory and buffers. Memory objects can be stored on different memory regions. Global memory is accessible by the host and kernels, while local and private memory don't. Objects may be stored as buffers or images, and they can have different read/write permissions.

Synchronization. Barriers may be created to synchronize computations and memory operations. Usually, homogeneous computations is the main objective, but against divergences in the code, barriers are used as a synchronization mechanism. Consistency memory reads and updates may require creating barriers over global or local memory.

Compute hierarchy

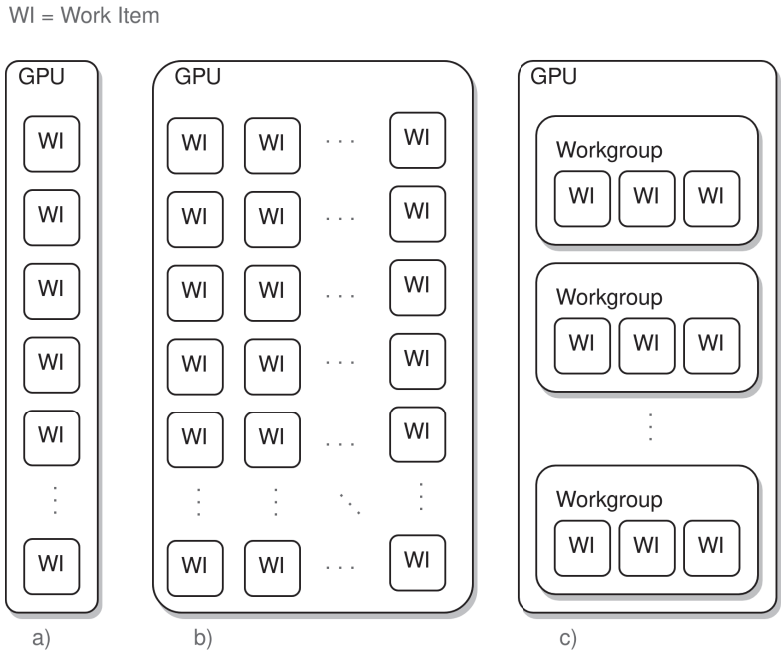


Figure 2.4: OpenCL compute hierarchy — Example of three different ways of arranging Compute Units (CU): a) 1D array. b) 2D array. c) In workgroups.

An execution of an OpenCL programs occurs in the host and in a device. The host creates the context, queue and memory objects that are going to be used to communicate with the device.

In the device, the workload is distributed across work-items. Work-items can be grouped into a work-group to share memory accesses and synchronization.

A device distributes their work items as a 1-dimensional array, but OpenCL provides the capability to sort them as 2D, 3D or N-Dimensional arrays. In case that a command execution exceeds the dimensions/work-items available, OpenCL partitions the execution, but this may have an impact in the overall performance. Each device can provide information for best fitting dimensions and work-group sizes.

Memory model

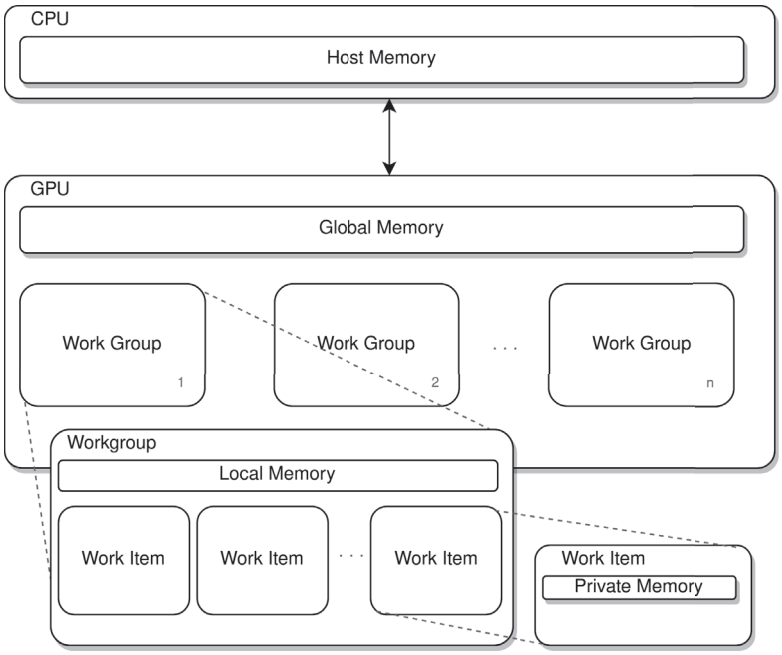


Figure 2.5: OpenCL memory model

OpenCL provides a protocol to communicate between host memory and device's memory, which may involve copying the objects or using a shared virtual memory. Device Memory is the one directly available to the kernels. Device Memory has three disjoint levels: global memory, local memory and private memory (Figure 2.5).

- Global memory can be read and written by host, kernels and shared across many devices. It's the biggest among all, but also the slowest one.
- Local memory is available to kernels and work-groups, with a maximum size much smaller than global. Kernels in the same work-group share this memory region and they could need synchronization

- Private memory is the fastest kind of memory in OpenCL. It corresponds to kernel's registers where variables are stored and it is not visible to other kernels.
- Outside the management of the programmer, devices could have dedicated caches to improve efficiency.

In this regions, OpenCL can store three types of objects: buffers, images and pipes. The most common one for us is the buffer, used to store general purpose values in a dense and sequential memory.

The host is the one than can dynamically allocate this objects, while kernels are tied to an static allocation model. The only case when kernels can allocate memory is at the creation of a child kernel, but they cannot access this memory.

This allocation policies will vastly influence the algorithms and computation model, as it is not equivalent to multi-core programming on CPUs.

2.2.2. Platform integration

OpenCL is designed to be very similar to C and C++, and they provide a library API in these languages, but integration is not tied to them only. Many other languages have libraries wrapping/binding this C/C++ APIs: Python, Java, Erlang, but even Haskell or Racket.

Compute Libraries

Many libraries proliferate in the ecosystem of GPGPU computing. Creating fine-tuned parallel algorithms is not an easy task, and sometimes, it can also be repetitive. As in many other disciplines, libraries for common computations are designed and published for ease of use. Their main advantage is portability across platforms: providing a common interface for multiple implementations and even sequential versions. Sometimes the handiness they provide may have a trade-off in overall performance, but an easy to parallelize algorithm can gain a boost in its performance when using these libraries.

Boost Compute is an C++ OpenCL library, designed to provide an easy interface similar to the standard C++ library and build on top of simple parallel operations. It provides fast parallel versions of high level operations like mapping, filtering and reducing algorithms.

Chapter 3

Proposal — Parallel ANF-SAT Solving

In this section we explain all the implementation details and the multiple iterations of the implementation. Each version had a specific goal, but all of them sharing the same base logic. With each step we change a component until we reach our more efficient version in a complete GPU parallel context.

Our first approach will be our baseline for parsing and algorithm validation. This version sets the common formats and interfaces to work with the rest of implementation and helps to build regression tests to check new iterations.

3.1. Sequential Implementation

This implementation is made in Haskell. One of the features that prevails in the core of the tool is the parsing step. We convert from multiple formats to our internal representation. We support DIMACS and a subset of SMT-LIB.

DIMACS support is oriented to easily work with common benchmarks made for SAT solvers. SMT-LIB provides a richer syntax to express complex formulas without the need of simplifying it to CNF as in DIMACS. Both formats needs to be converted to ANF. We only support a subset of SMT-LIB that works with first order logic with these operators:

- Boolean variables
- Negation
- Or
- And
- Implication

- Double implication

3.1.1. Conversion to ANF

These two encoding are converted to an internal data type that support these operations. Once we have our syntax tree, we start to apply transformations to obtain an ANF formula (sec 2.1.1).

Each formula is converted to canonical ANF. But we don't convert the whole file directly, as this would be equivalent to a solver. The conversion algorithm is only applied to sub-formulas.

3.1.2. Abstract encoding

One of the simplest encodings that we can do is a direct conversion to an AST. Each monomial in a canonical ANF can be encoded as a list of boolean variables, all of them associated with the AND operator.

$$x \cdot y \cdot z \mapsto [x, y, z]$$

On the higher level we find a collection of XORs. We can nest two arrays to encode our canonical ANF.

$$xyz \oplus xz \mapsto [[x, y, z], [x, z]]$$

As Haskell data with type wrappers:

```
XOr [ And [Var 'x', Var 'y', Var 'z' ]
    , And [Var 'x', Var 'z' ]
    ]
```

A particularity of this data type is that we enforce a correct construction of the ANF, as we cannot mix levels of conjunctions.

```
data ANF a = ANF (XOr a)
data XOr a = XOr [And a]
data And a = And [Var a]
data Var a = Var a | One
```

3.1.3. Solving by ANF simplification

With this first encoding we can start our first solver. This solver is sequential, without supporting any kind of parallelism. It will be very inefficient, as we are working with abstract encodings, but in the core of this solver we find the same principles that later we are going to use.

$$m_1 = x_1x_2x_3 \qquad m_3 = x_1x_2 \qquad m_5 = x_3x_4$$
$$m_2 = x_1x_3 \qquad m_4 = x_1$$

$$\begin{aligned}(m_1 \oplus m_2) \cdot (m_3 \oplus m_4 \oplus m_5) = \\ m_1 \cdot (m_3 \oplus m_4 \oplus m_5) \oplus m_2 \cdot (m_3 \oplus m_4 \oplus m_5) = \\ x_1x_2x_3x_1x_2 \oplus x_1x_2x_3x_1 \oplus x_1x_2x_3x_3x_4 \oplus x_1x_3x_1x_2 \oplus x_1x_3x_1 \oplus x_1x_3x_3x_4 = \\ x_1x_2x_3 \oplus x_1x_2x_3 \oplus x_1x_2x_3x_4 \oplus x_1x_2x_3 \oplus x_1x_3 \oplus x_1x_4 = \\ x_1x_2x_3x_4 \oplus x_1x_2x_3 \oplus x_1x_3 \oplus x_1x_4 = \\ x_1x_3 \oplus x_1x_4 \oplus x_1x_2x_3 \oplus x_1x_2x_3x_4\end{aligned}$$

We perform a sequential polynomial multiplication. The resulting polynomial has an initial size of nxm, but we need to reduce it until we obtain the canonical form. With this encoding we can use the same reduction mechanism as in the formula parsing and transformation step.

The solver continues by taking this new reduced formula and starting again the process with the next input. At the moment we obtain and empty formula or when we run out of new inputs, the solver terminates. This iterative process is common to all solvers.

3.2. Sequential Vector Implementation

In the next implementation we improved the internal encoding, working with an efficient implementation in terms of algebraic operations. We keep the same steps of parsing and reducing, but once we have our clause in canonical ANF, we apply the encoding described in Section 2.1.3.

For an initial expression in SMT-LIB format, we obtain the following ANF:

```
(assert (or (not x_1) (not x_2)))
```

$$(x_1 \oplus 1)(x_2 \oplus 1) \oplus x_1 \oplus 1 \oplus x_2 \oplus 1$$

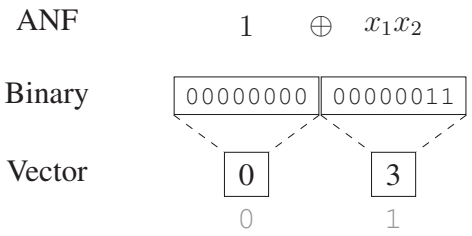
We need to apply the reduction to a canonical ANF:

$$\begin{aligned} & x_2(x_1 \oplus 1) \oplus 1(x_1 \oplus 1) \oplus x_1 \oplus 1 \oplus x_2 \oplus 1 \\ =& x_1x_2 \oplus 1 \cdot x_2 \oplus 1 \cdot x_1 \oplus 1 \cdot 1 \oplus x_1 \oplus 1 \oplus x_2 \oplus 1 \\ =& 1 \oplus x_1x_2 \end{aligned}$$

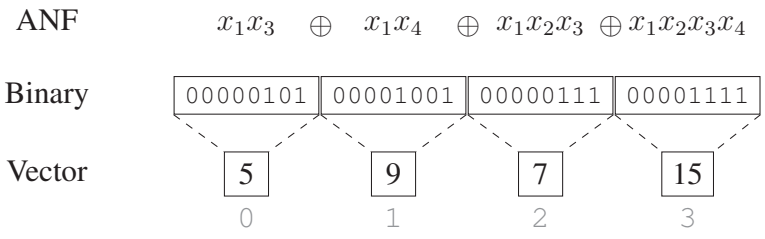
Once we have a canonical ANF we start the encoding process. We do not have many variables declared yet, an 8-bits word could be representative enough for this example. Each variable parsed will have a unique index assigned:

Variable	Index	(8 bit word)
x_1	0	`00000001`
x_2	1	`00000010`

The expression $1 \oplus v1_1v1_2$ will be encoded to a length 2 with the monomials.



We take the previously parsed and reduced formula to obtain our second polynomial in vector encoding:



3.2.1. Vector Multiplication

In order to multiply each monomial in this encoding, we need to apply the bit level "or" (|), as described in Section 2.1.3.1. The sequential implementation in Haskell is as follows:

```
vAnd :: (FiniteBits a, Storable a) => Vector a -> Vector a -> Vector a
vAnd v1 v2 =
  let l1 = V.length v1
      l2 = V.length v2
      l' = l1 * l2
      core i =
          let a = i `mod` l1
              b = i `div` l1
          in  (v1 ! a) .|. (v2 ! b)
  in  Vector.generate l' core
```

In this implementation we can see the potential of parallelization. We generate a new polynomial were each monomial can be computed independently.

		x_1x_3	x_1x_4	$x_1x_2x_3$	$x_1x_2x_3x_4$
		00000101	00001001	00000111	00001111
1	00000000	00000101	00001001	00000111	00001111
x_1x_2	00000011	00000111	00001011	00000111	00001111

The final vector of integers is:

3	9	7	15	7	11	7	15

3.2.2. Set Reduction

Once we have the resulting polynomial we need to reduce it. In this implementation the reduction step is done with a int set construction:

```
reduce :: (Integral a, Storable a) => Vector a -> Vector a
reduce = toVector . V.foldl' addMonom IntS.empty
```

The final formula can be decoded back:

(AND)	x_1x_3	x_1x_4	$x_1x_2x_3$	$x_1x_2x_3x_4$
1	x_1x_3	x_1x_4	$x_1x_2x_3$	$x_1x_2x_3x_4$
x_1x_2	$x_1x_2x_3$	$x_1x_2x_4$	$x_1x_2x_3$	$x_1x_2x_3x_4$

$$x_1x_3 \oplus x_1x_4 \oplus x_1x_2x_3 \oplus x_1x_2x_3x_4 \oplus x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_1x_2x_3x_4$$
$$= x_1x_3 \oplus x_1x_4 \oplus x_1x_2x_3 \oplus \oplus x_1x_2x_4 \oplus x_1x_2x_3x_4$$

3.3. GPU implementation

In this implementation we finally rewrite our previous versions in a parallel context. The implementation of the multiplication nearly correlates one to one with the previous presented. Reduction, on the other hand, has a complete new approach. We could preserve our sequential reduction via set construction, but we will suffer from communication delay between CPU-GPU and higher memory usage.

Work-items distribution

The kernel that we are going to use receives three data structures as arrays:

1. Accumulated ANF;
2. New ANF to multiply;
3. Resulting unreduced ANF.

In the naive implementation of the kernel, each work item will have an index corresponding with a position in the output version. From this index we can compute the corresponding pair of monomials that we need to take and perform the AND operation.

```
  /* Scalar Product - a(b + c) = ab + ac */
__kernel void anfAND(int anf_1_size,   const __global uint *anf_1,
                     int anf_2_size,   const __global uint *anf_2,
                     int out_anf_size, __global uint *out_anf){

  int gid = get_global_id(0);

  if (gid < out_anf_size) {
    int anf_1_i = gid % anf_1_size;
    int anf_2_i = gid / anf_1_size;
    // It's an and operation, but merging variables is an or
    out_anf[gid] = anf_1[anf_1_i] | anf_2[anf_2_i];
  }
}
```

Kernels property with respect to OpenCL implementations

We could tune this implementation to work with specific sizes of work groups, but the compiler should notice this code approach and optimize equally for it.

Memory access. This kernel only access to global memory, with the input vectors in the region of `const` memory. Work-items are not divided into work-groups, but sequential

work-items access to contiguous elements in the first kernel.

Work distribution by Compute Unit. Each work-item is not tied to a work-group, but the OpenCL platform will fit as many individual work-items in a compute-unit, as we are not restricting the dimensions to work with. Each device has a limit size of Compute Units and work-items inside them, any exceeding amount of work-items will be queued waiting to the first round of work-items to finish. p

Vectorization. This kernel shown is not using any vectorization capabilities of the hardware, which could improve dense memory access and full usage of the work-item's ALU.

Equivalent sequential implementation. At the end, we have just implemented without further optimizations the function `[a' .|. b' | a' <- a, b' <- b]`.

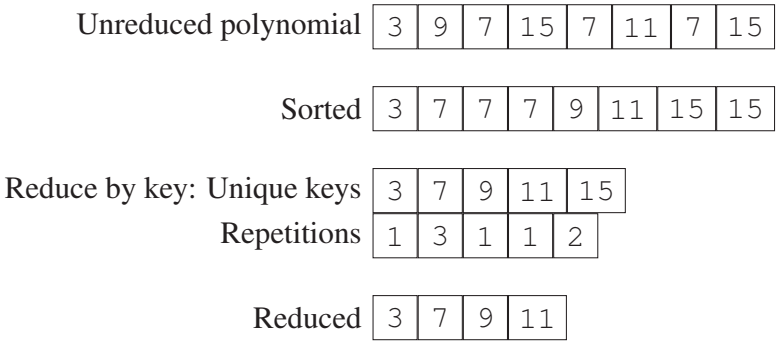
Parallel Reduction

Reduction steps are not usually efficient operations in GPUs. Multiple data dependencies and high memory usage can have a harmful impact on the algorithm speed. But, on the other hand, data communication between CPU and GPU can also provoke an overhead, specifically when generated data can grow exponentially. In this scenario, even a simple parallel reduction in the GPU can outperform the sequential implementation.

Keys ideas in the algorithm:

1. Sort resulting vector.
2. Reduce by key.
3. Remove values repeated an even number of times.

For the first two steps we rely on the OpenCL C++ library Boost Compute [27], which provides implementations of these operations adjusted to GPU specific sized. Sorting is done in local groups with a radix sort. Reducing by key is computed as with a local scan technique, where the values of the array are combined with a custom operation. In this case, two arrays are generated, one with the unique keys and a second one with the number of repetitions. Finally, we copy to a new array only the values repeated an odd number of times.



The last operation is performed with a similar model and partially extracted from key building block in this OpenCL library. These steps take advantage of already existent objects in the GPU, improving data reusing and avoiding copying overhead.

Memory reuse

During the whole process of multiplication/reduction, computed vector remain in the GPU memory. We only feed and copy new parsed formulas into the GPU. In the process of reducing the vector we can make checks about the length of the reduced polynomial to check unsatisfiable cases.

Conditions of Unsatisfiability

When the reduced vector is empty, the formula has turned UNSAT and execution should stop. No more clauses should be consumed. If the reduced vector is not empty, we repeat the algorithm until consuming all the assertions.

3.4. Algorithm Validation

In order to check if our implementation is giving correct results, we have two approximations: a regression test suite with previous solvers and an oracle approach with an external solver.

We generate test with custom data generators using QuickCheck, a property based testing tool. These generated boolean formulas are converted into a common input format for all solvers: SMT-LIB.

Oracle test suite

We use an external solver, Z3, to check that our results and conditions of unsatisfiability are at least equivalent to what we assume a correct implementation.

Regression test suite

Once we have a validated a base algorithm, we can apply the same generated approach to check equivalence between reduced ANF formulas. No matter how the are computed, the canonical form is unique, so solvers must reach the same solution.

Chapter 4

Experimental Results

In this chapter we are going to describe the outcomes of the tests suites, performance tests and limitations seen.

We have developed a tool that can check satisfiability of an ANF sat formula. It can run on different implementations:

- Sequential Haskell.
- Sequential Haskell and parallel OpenCL multiplication.
- Full parallel C++ and OpenCL.

4.1. Setup

Code for this implementations can be found in Appendix A, but it is also publicly available at GitHub[1].

Testing machine

All tests have been performed in a desktop computer with an AMD Ryzen 7 3700X 8-Core Processor, 16GB of DDR4 RAM at 3200MHz. The graphics card is a Radeon R9 380 with 4GB of dedicated memory.

[1] `https://github.com/ignaciobll/ANF-`

Build tools

In order to build the code we need to have the OpenCL headers installed, the `stack` tool for Haskell[2] and `g++` for the C++ implementation.

Test suites

Test suites can be run executing ‘stack test‘ and should outcome a trace similar to this one:

```
SATCL
  Steps
    All steps are equivalent: Sequential == Parallel
      +++ OK, passed 20 tests.
  Sequential.
    is equivalent to Z3 oracle
      +++ OK, passed 100 tests.
  Parallel.
    is equivalent to Z3 oracle
      +++ OK, passed 20 tests.
```

In the trace we can see how only 20 test are run on parallel test suite because the later the test, the bigger it is. When running big tests at the same time in the OpenCL runtime it can generate some memory issues. Out of the scope of algorithm validation.

4.2. Algebraic Normal Form size

This algorithm may potentially blow up in memory usage. Theoretically, polynomial multiplication could grow exponentially if monomials are completely independent in their variables. We assume that for any realistic problem this would not be the case. In this study we compare the grow of an actual SAT problem and compare it with their theoretical upper bound.

In Figure 4.1 we can see how at the first multiplications the polynomial size grows exponential at the same rate as the upper bound. Some iterations after the polynomial seems to reach a limit and starts lowly to decrease its size.

With this results we can say that at least we have improved over an algorithm that is brute force, as some early implementations of GPU SAT solvers tried.

[2] `https://docs.haskellstack.org/en/stable/README/`

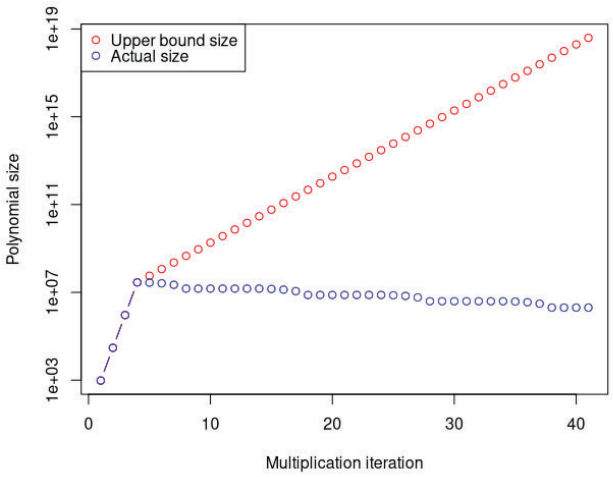


Figure 4.1: Polynomial length upper bound vs actual size per iteration (log scale)

4.3. Sequential vs Parallel times

One of the first things that we notice when we run the parallel implementation is big startup time. This is because kernel loading and compilation to the target platforms requires some time. After the first usage, kernel binaries are already on GPU memory. These first overhead can be solved partially by pre-compiling common used kernels and loading them at program startup. Unfortunately, not all kernels can benefit from these approach as some of them are dynamically compiled based on input size.

If we take a look at the program output when running the parallel implementation we can see a context preparation of common kernels. Also, first iterations take so much time compared to the next ones even though ANF sizes are smaller.

```
Time spent building reduce.cl: 489 ms
Preparing boost context... 1325 ms
iter 0: 1850577 µs (gpu) [...]
iter 6:    7447 µs (gpu)
```

Deviation can observed in Table 4.1 and Figure

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	Std. dev.
2890	2956	3305	61671	6336	1884599	292565

Table 4.1: Time distribution in a parallel execution in μs

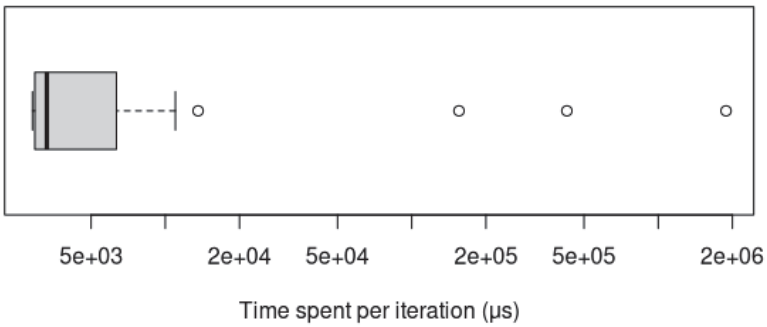


Figure 4.2: Time spent per parallel iteration in μs (log scale)

It's important to notice that we also include a little overhead when copying data from main memory to GPU. This time oscillate around 300 μs each iteration.

Once we know all this peculiarities of the parallel runtime, we can compare with the sequential version how it behaves. In Figure 4.3 we can see how our parallel implementation is a couple of orders of magnitude faster than the sequential one.

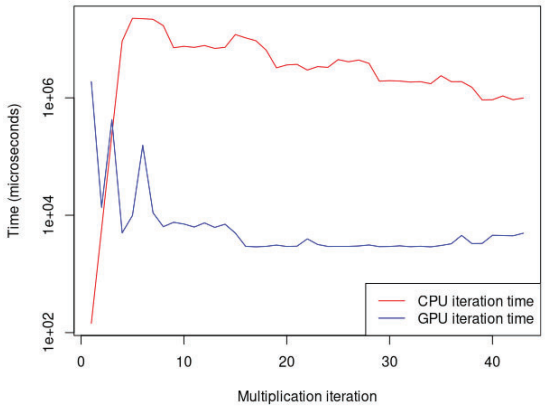


Figure 4.3: Iteration time: GPU vs CPU (log scale)

4.4. GPU profiling

We have take a deeper look at the execution times inside the GPU kernels. The distribution of time is proportional in every iteration, with a sample shown in Figure 4.4

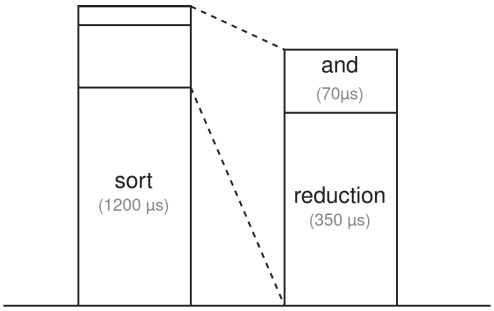


Figure 4.4: Kernel mean execution times

This is just the time spent on kernel execution, but there is some extra time needed for memory management, copying data and between CPU and GPU and the runtime synchronization.

The results show that the reduction step, in which we may include the sorting call, takes a huge proportion of the execution time. In particular, the `sort` call, implemented and optimized by the *Boost* library, is the single kernel execution that results in the majority of time. As expected, the multiplication of polynomials is a very efficient operation, considering that this call time includes the initialization and synchronization steps.

4.5. Comparison with other solvers

Although the objective of this study is not beating current SAT solvers, we perform a little comparison with Z3, for completeness and drawing better the picture of solvers execution times. This tool doesn't end in a good position when compared to state of the art solvers. It has even problems and execution errors because of memory faults, without being able to finish some commonly used benchmarks in DIMACS.

We have compared some execution times with Z3 solver, with results show in Table 4.2.

Problem name	Z3	*Our Tool* (Anfeta)
`uf20-01.cnf`	3ms	1507ms + 1498ms of initialization
`uf20-20.cnf`	3ms	1523ms + 1531ms of initialization

Table 4.2: Time results compared with Z3 solver

We don't need to examine many tests to see the huge difference between them.

Chapter 5

Discussion and conclusions

5.1. Limitations

During the experiments we have encountered several limitations in this solver.

Number of variables. Our encoding limits the number of variables that we can handle. By default, we can work with a maximum of 64 variables, as it is the word limit, and we represent each variable with a bit. We could go beyond this limit if we start to work with vector data type, which depending on the architecture it could handle up to 512 bits. The problem with this upgrade is that it will also affect the performance, because we could fit fewer monomials in the same memory size polynomial. State of the Art solvers can work efficiently with thousands or even millions of variables.

Memory usage. Although we have seen how polynomial size can stop increasing its size indefinitely, it is near an exponential growth depending on specific problem characteristics. The size in which it stabilizes could be impractical to work, creating slow iterations even on GPU. For the most extreme cases, it can even surpass the memory limit in the GPU, ending with an error in the execution.

Speed. Parallel implementation can boost the execution several orders of magnitude respect to the sequential implementation, but it cannot be compared in performance with many solvers. Even compared with sequential solvers it is impractical and may suffer solving even simple problems. Currently, it is far away from other algorithms.

5.1.1. Possible overcome of limitations

In future work we could specialize our solver to work on specific sets of variables. We could try to partition the formula to work on smaller chunks minimizing the number of variables. We will require new steps to finally merge the formula, but depending on the problem, this technique could have a positive impact.

Memory usage could also be benefited of the same partition approach. Smaller formulas could tend to grow less. If we partition or reorder formulas to have more independent variables, multiplication and simplification could be done keeping controlled their sizes and eventually merging those sub-formulas.

The GPU hardware used in the tests have some antiquity. It is a powerful device with really high parallelism capabilities, but current models have improved a lot in terms of memory and compute units. This execution speed is far to be on par with CPU algorithms, but CPU performance increase is less that GPUs are experiencing. The potential of joining several GPU devices should be considered, with even more attention if a partition algorithm is implemented.

A possible drawback that a partition algorithm like this could have is the delay in detecting unsatisfiable instances. These problems should be addressed as soon as possible, because with the current approach we cannot go back in the reduction steps.

5.2. Challenges during development

Setting up the environment to properly test a GPU program may not be a trivial task. Drivers and compatibility issues have played a major role during early stages of development. Usually, tools commonly used are mostly available to newer version of graphics cards, while older tools that support used models aren't supported any more.

Current state of library support is difficult to obtain, and in several occasions fixes have been applied in third party libraries. In some cases to completely fix the lib and being able to uses it and in other cases to adapt it to our needs. This work is of course less than implementing them from scratch, but in any case, it is a challenge to be addressed during the research and development.

We would have like to try and build on top of the previous solver made by David Lilue, but we could get a complete copy of the code to make it work. Some time was spent trying to adapt it, but slightly different decisions in the implementation ended with a new implementation from scratch. Fortunately, we could be able to overcome some of the limitations and future work that was discussed in that work. We could, for example, have a full GPU solver in C++, a requisite stated in their work.

Chapter 6

Bibliography

[1] Gilles Audemard and Laurent Simon. “Lazy Clause Exchange Policy for Parallel SAT Solvers”. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Cham: Springer International Publishing, 2014, pp. 197–205. ISBN: 978-3-319-09284-3.

[2] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI’09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[3] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, 2009. ISBN: 1586039296.

[4] Armin Biere. “Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013”. In: 2013.

[5] Armin Biere et al. “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020”. In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[6] Davin Choo et al. “Bosphorus: Bridging ANF and CNF Solvers”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2019), pp. 468–473.

[7] Geoffrey Chu, Peter J. Stuckey, and Aaron Harwood. *PMiniSAT: A Parallelization of MiniSAT 2.0*. Tech. rep. 2008.

[8] C. B. E. Costa. “Parallelization of SAT Algorithms on GPUs”. In: 2014.

[9] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[10] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

[11] Jorge Fernandez Davil. "Zhegalkin Polynomial SAT Solver". MA thesis. University of Knet, 2017.

[12] Johannes Klaus Fichte, Markus Hecher, and Stefan Szeider. "A Time Leap Challenge for SAT Solving". In: *CoRR* abs/2008.02215 (2020). arXiv: 2008.02215. URL: https://arxiv.org/abs/2008.02215.

[13] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver". In: *JSAT* 6 (June 2009), pp. 245–262. DOI: 10.3233/SAT190070.

[14] Youssef Hamadi and Christoph Wintersteiger. "Seven Challenges in Parallel SAT Solving". In: *AI Magazine* 34.2 (June 2013), p. 99. DOI: 10.1609/aimag.v34i2.2450. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2450.

[15] Nguyen Hung. "Combinations of Boolean Gröbner Bases and SAT Solvers". PhD thesis. Dec. 2014. DOI: 10.13140/RG.2.2.25392.92167.

[16] Matti Järvisalo et al. "The International SAT Solver Competitions". In: *AI Magazine* 33.1 (Mar. 2012), pp. 89–92. DOI: 10.1609/aimag.v33i1.2395. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2395.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134397606.

[18] Ludovic Le Frioux et al. "Modular and Efficient Divide-and-Conquer SAT Solver on Top of the Painless Framework". In: *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*. Vol. 11427. Lecture Notes in Computer Science. Springer, Cham, Apr. 2019, pp. 135–151.

[19] Ludovic Le Frioux et al. "PaInleSS: a Framework for Parallel SAT Solving." In: *The 20th International Conference on Theory and Applications of Satisfiability Testing*. Vol. 10491. Lecture Notes in Computer Science. Melbourne, Australia: Springer, Aug. 2017, pp. 233–250. DOI: 10.1007/978-3-319-66263-3_15. URL: https://hal.archives-ouvertes.fr/hal-01540785.

[20] Jia Liang et al. "Learning Rate Based Branching Heuristic for SAT Solvers". In: July 2016, pp. 123–140. ISBN: 978-3-319-40969-6. DOI: 10.1007/978-3-319-40970-2_9.

[21] Quirin Meyer et al. "3-SAT on CUDA: Towards a massively parallel SAT solver". In: Aug. 2010, pp. 306–313. DOI: 10.1109/HPCS.2010.5547116.

[22] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: Association for Computing Machinery, 2001, pp. 530–535. ISBN: 1581132972. DOI: 10.1145/378239.379017. URL: https://doi.org/10.1145/378239.379017.

[23] Muhammad Osama and Anton Wijs. “Parallel SAT Simplification on GPU Architectures”. In: Apr. 2019, pp. 21–40. ISBN: 978-3-030-17461-3. DOI: 10.1007/978-3-030-17462-0_2.

[24] Alessandro Dal Palù et al. “CUD@SAT: SAT solving on GPUs”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 27.3 (2015), pp. 293–316. DOI: 10.1080/0952813X.2014.954274. eprint: https://doi.org/10.1080/0952813X.2014.954274. URL: https://doi.org/10.1080/0952813X.2014.954274.

[25] João P. Marques Silva and Karem A. Sakallah. “GRASP—a New Search Algorithm for Satisfiability”. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’96. San Jose, California, USA: IEEE Computer Society, 1997, pp. 220–227. ISBN: 0818675977.

[26] Ali Asgar Sohanghpurwala, Mohamed W. Hassan, and Peter Athanas. “Hardware accelerated SAT solvers—A survey”. In: *Journal of Parallel and Distributed Computing* 106 (2017), pp. 170–184. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2016.12.014. URL: https://www.sciencedirect.com/science/article/pii/S0743731516301903.

[27] Jakub Szuppe. “Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL”. In: *Proceedings of the 4th International Workshop on OpenCL*. IWOCL ’16. Vienna, Austria: Association for Computing Machinery, 2016. ISBN: 9781450343381. DOI: 10.1145/2909437.2909454. URL: https://doi.org/10.1145/2909437.2909454.

Appendix A

Code

A.1. C++ implementation

A.1.1. OpenCL Kernels

```
/* Scalar/Polynomial Product - a(b + c) = ab + ac */
__kernel void anf_and(int anf_1_size,   const __global uint *anf_1,
                      int anf_2_size,   const __global uint *anf_2,
                      int out_anf_size, __global uint *out_anf) {
  int gid = get_global_id(0);
  if (gid < out_anf_size) {
    int anf_1_i = gid % anf_1_size;
    int anf_2_i = gid / anf_1_size;
    // It's an and operation, but merging variables is an or
    out_anf[gid] = anf_1[anf_1_i] | anf_2[anf_2_i];
  }
}

__kernel void odd_indices(const __global uint *values, __global uint *indices)
{
  const uint gid = get_global_id(0);
  indices[gid] = values[gid] % 2 != 0 ? 1 : 0;
}

__kernel void local_scan(__global uint* block_sums,
                         __local uint* scratch,
                         const uint block_size,
                         const uint count,
                         const uint init,
                         __global uint* _buf0, // Read indices
                         __global uint* _buf1) // Write indices
{
```

```
28    const uint gid = get_global_id(0);
29    const uint lid = get_local_id(0);
30
31    if(gid < count){
32      // When exclusive scan:
33      /*
34      const uint local_init = (gid == 0) ? init : 0;
35
36      if(lid == 0){
37        scratch[lid] = local_init;
38      } else {
39        scratch[lid] = _buf0[gid-1];
40      }
41      */
42      scratch[lid] = _buf0[gid-1];
43    } else {
44      scratch[lid] = 0;
45    }
46    barrier(CLK_LOCAL_MEM_FENCE);
47
48    for(uint i = 1; i < block_size; i <<= 1){
49      const uint x = lid >= i ? scratch[lid-i] : 0;
50      barrier(CLK_LOCAL_MEM_FENCE);
51      if(lid >= i){
52        scratch[lid] = ((scratch[lid])+(x));
53      }
54      barrier(CLK_LOCAL_MEM_FENCE);
55    }
56
57    if(gid < count){
58      _buf1[gid] = scratch[lid];
59    }
60
61    if(lid == block_size - 1 /* && gid < count */) {
62      block_sums[get_group_id(0)] = scratch[lid];
63    }
64  }
65
66  __kernel void write_scanned_output(__global uint* output,
67                                     const __global uint* block_sums,
68                                     const uint count)
69  {
70    const uint gid = get_global_id(0);
71    const uint block_id = get_group_id(0) + 1;
72
73    if (gid < count) {
74      output[gid] = block_sums[block_id] + output[gid];
75    }
76  }
```

```

__kernel void copy_reduced_indices(const __global uint * indices,
                                   const __global uint * keys,
                                   const __global uint * values,
                                   __global uint * v)
{
  const uint gid = get_global_id(0);

  if (values[gid] % 2 != 0) {
    v[indices[gid]] = keys[gid];
  }
}
```

A.1.2. Main Program (CPU)

```
#include <vector>
#include <algorithm>
#include <boost/compute.hpp>
#include <chrono>
#include <sstream>
#include <iostream>
#include <string>
#include <iomanip>

namespace compute = boost::compute;

void prepareBoostComputeKernels(compute::context ctx,
                                compute::command_queue queue) {

  std::cout << "Preparing boost context... ";
  auto t1 = std::chrono::steady_clock::now();

  compute::vector<int> insertion(10, ctx);
  compute::vector<int> radix(100, ctx);
  compute::vector<int> radix_values(100, ctx);
  compute::vector<int> out_values(100, ctx);
  compute::vector<int> out_keys(100, ctx);

  compute::sort(insertion.begin(), insertion.end(), queue);
  compute::sort(radix.begin(), radix.end(), queue);

  compute::fill(radix_values.begin(), radix_values.end(), 1, queue);

  compute::reduce_by_key(radix.begin(), radix.end(), radix_values.begin(),
                         out_keys.begin(), out_keys.end(), queue);

  auto t2 = std::chrono::steady_clock::now();

  std::cout
```

```
35      << std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count()
36      << " ms" << std::endl;
37
38  }
39
40  std::vector<std::vector<int>> parseProgramStdIn() {
41
42    std::vector<std::vector<int>> program;
43
44    for (std::string line; std::getline(std::cin, line);) {
45      std::vector<int> tmp;
46
47      std::istringstream lineS(line);
48      int n;
49
50      while (lineS >> n){
51        tmp.push_back(n);
52      }
53
54      if (tmp.size() > 0)
55        program.push_back(tmp);
56    }
57
58    return program;
59  }
60
61  inline bool file_exists (const std::string& name) {
62      return ( access( name.c_str(), F_OK ) != -1 );
63  }
64
65  void writeFileBytes(const char* filename,
66                      std::vector<unsigned char>& fileBytes){
67      std::ofstream file(filename, std::ios::out|std::ios::binary);
68      std::copy(fileBytes.cbegin(), fileBytes.cend(),
69          std::ostream_iterator<unsigned char>(file));
70  }
71
72
73  void printProgram(std::vector<std::vector<int>> program) {
74    for (auto anf = program.begin(); anf != program.end(); ++anf) {
75      for (auto monom = anf->begin(); monom != anf->end(); ++monom) {
76        std::cout << *monom << ' ';
77      }
78      std::cout << '\n';
79    }
80  }
81
82  template<class T>
83  void printVector(std::vector<T> v, std::string name = "", size_t length = -1)
```

```
{
  if (name.compare("")){
    std::cout << name << '\t';
  }
  if (length == -1) {
    length = v.size();
  }
  std::cout << "[ ";
  for (size_t i = 0; i < length; ++i)
    std::cout << std::setw(3) << v[i] << ' ';
  std::cout << "]" << std::endl;
}

inline size_t pick_scan_block_size(size_t size)
{
    if(size == 0)         { return 0; }
    else if(size <= 1)    { return 1; }
    else if(size <= 2)    { return 2; }
    else if(size <= 4)    { return 4; }
    else if(size <= 8)    { return 8; }
    else if(size <= 16)   { return 16; }
    else if(size <= 32)   { return 32; }
    else if(size <= 64)   { return 64; }
    else if(size <= 128)  { return 128; }
    else                   { return 256; }
}

compute::vector<int> anf_product_reduce(compute::vector<int> a,
                                        compute::vector<int> b,
                                        compute::program program,
                                        compute::context ctx,
                                        compute::command_queue queue)
{
  compute::kernel anf_andK(program, "anf_and");
  compute::kernel odd_indicesK(program, "odd_indices");
  compute::kernel local_scanK(program, "local_scan");
  compute::kernel write_scanned_outputK(program, "write_scanned_output");
  compute::kernel copy_reduced_indicesK(program, "copy_reduced_indices");

  /* 1. AND */

  compute::vector<int> c(a.size() * b.size(), ctx);

  anf_andK.set_arg(0, static_cast<cl_int>(a.size()));
  anf_andK.set_arg(1, a);
  anf_andK.set_arg(2, static_cast<cl_int>(b.size()));
  anf_andK.set_arg(3, b);
  anf_andK.set_arg(4, static_cast<cl_int>(c.size()));
  anf_andK.set_arg(5, c);
```

Code

```
133
134         compute::event event_anf_andK = queue.enqueue_1d_range_kernel(anf_andK, 0, c.size
135
136         event_anf_andK.wait();
137         std::cout << "PROFILE anf_andK: " << event_anf_andK.duration<std::chrono::micros
138
139   /* 2. sort */
140
141
142         auto t1_sort = std::chrono::steady_clock::now();
143   compute::sort(c.begin(), c.end(), queue);
144         auto t2_sort = std::chrono::steady_clock::now();
145         auto duration_sort = std::chrono::duration_cast<std::chrono::microseconds>(t2_so
146         std::cout << "PROFILE sort: " << duration_sort << " µs" << std::endl;
147
148   /* 3. Reduce by key */
149
150   compute::vector<int> c_keys(c.size(), ctx);
151   compute::vector<int> c_values(c.size(), ctx);
152   compute::vector<int> c_ones(c.size(), ctx);
153   compute::fill(c_ones.begin(), c_ones.end(), 1, queue);
154
155   auto kv_pair_size = compute::reduce_by_key(c.begin(), c.end(), c_ones.begin(),
156                                              c_keys.begin(), c_values.begin(), queue);
157
158   size_t kv_size = std::distance(c_keys.begin(), kv_pair_size.first);
159
160   /* 4. Indices */
161
162   compute::vector<uint> indices(kv_size, ctx);
163
164   odd_indicesK.set_arg(0, c_values);
165   odd_indicesK.set_arg(1, indices);
166
167         compute::event event_odd_indices = queue.enqueue_1d_range_kernel(odd_indicesK, 0
168
169   event_odd_indices.wait();
170         std::cout << "PROFILE odd_indices: " << event_odd_indices.duration<std::chrono::n
171
172   size_t last_mask_element = (indices.cend() - 1).read(queue);
173
174   /* 4.1 Scan indices (inclusive) */
175
176   size_t block_size = pick_scan_block_size(kv_size);
177   size_t block_count = kv_size / block_size;
178
179   if (block_count * block_size < kv_size) {
180     block_count++;
181   }
```

```
182
183   compute::vector<int> block_sums(block_count, ctx);
184   compute::fill(block_sums.begin(), block_sums.end(), 0, queue);
185
186   local_scanK.set_arg(0, block_sums);
187   local_scanK.set_arg(1, compute::local_buffer<cl_uint>(block_size));
188   local_scanK.set_arg(2, static_cast<cl_uint>(block_size));
189   local_scanK.set_arg(3, static_cast<cl_uint>(kv_size));
190   local_scanK.set_arg(4, static_cast<cl_uint>(0));
191   local_scanK.set_arg(5, indices);
192   local_scanK.set_arg(6, indices);
193
194         compute::event event_local_scanK = queue.enqueue_1d_range_kernel(local_scanK,
195
196
197
198
199         event_local_scanK.wait();
200         std::cout << "PROFILE local_scanK: " << event_local_scanK.duration<std::chrono::n
201
202   /* 4.2 Inclusive scan with block sums */
203
204   size_t block_size_2 = pick_scan_block_size(block_count);
205   size_t block_count_2 = kv_size / block_size;
206
207   if (block_count_2 * block_size_2 < block_count) {
208     block_count_2++;
209   }
210
211   compute::vector<int> block_sums_2(block_count_2, ctx);
212   compute::fill(block_sums_2.begin(), block_sums_2.end(), 0, queue);
213
214   local_scanK.set_arg(0, block_sums_2);
215   local_scanK.set_arg(1, compute::local_buffer<cl_uint>(block_size_2));
216   local_scanK.set_arg(2, static_cast<cl_uint>(block_size_2));
217   local_scanK.set_arg(3, static_cast<cl_uint>(block_count));
218   local_scanK.set_arg(4, static_cast<cl_uint>(0));
219   local_scanK.set_arg(5, block_sums);
220   local_scanK.set_arg(6, block_sums);
221
222         compute::event event_local_scanK2 = queue.enqueue_1d_range_kernel(local_scanK,
223                                 0,
224                                 block_count_2 * block_size_2,
225                                 block_size_2);
226
227   event_local_scanK2.wait();
228         std::cout << "PROFILE local_scanK2: " << event_local_scanK2.duration<std::chrono
229
230
```

Code

```
231   /* 4.3 Write with accumulated sum in blocks */
232
233   write_scanned_outputK.set_arg(0, indices);
234   write_scanned_outputK.set_arg(1, block_sums);
235   write_scanned_outputK.set_arg(2, static_cast<cl_uint>(kv_size));
236
237         compute::event event_write_scanned_outputK =
238                 queue.enqueue_1d_range_kernel(write_scanned_outputK,
239                                 block_size,
240                                 block_count * block_size,
241                                 block_size);
242
243   event_write_scanned_outputK.wait();
244         std::cout << "PROFILE write_scanned_outputK: " << event_write_scanned_outputK.du
245
246
247   /* 4.4 Size of reduced vector */
248
249   size_t out_size = last_mask_element + (indices.cend() - 1).read(queue);
250
251   /* 5. Copy indices to result */
252
253   compute::vector<int> out(out_size, ctx);
254
255   copy_reduced_indicesK.set_arg(0, indices);
256   copy_reduced_indicesK.set_arg(1, c_keys);
257   copy_reduced_indicesK.set_arg(2, c_values);
258   copy_reduced_indicesK.set_arg(3, out);
259
260
261         auto t1_copy_reduced = std::chrono::steady_clock::now();
262
263         compute::event event_copy_reduced_indicesK =
264                 queue.enqueue_1d_range_kernel(copy_reduced_indicesK, 0, indices.size(),
265
266   event_copy_reduced_indicesK.wait();
267         std::cout << "PROFILE copy_reduced_indicesK: " << event_copy_reduced_indicesK.du
268
269         auto t2_copy_reduced = std::chrono::steady_clock::now();
270         auto duration_copy_reduced = std::chrono::duration_cast<std::chrono::microsecond
271         std::cout << "PROFILE copy_reduced_indiceK (host): " << duration_copy_reduced <<
272
273
274   /* 6. END */
275
276   return out;
277 }
278
279 size_t anf_reduce_by_key_cpu (const std::vector<int> &keys,
```

```
                                std::vector<int> &keys_output,
                                std::vector<int> &values_output) {
  size_t size = 0;

  keys_output[0] = keys[0];
  values_output[0] = 1;
  for (size_t i = 1; i < keys.size(); ++i) {
    if (keys[i] != keys[i-1]) {
      ++size;
      keys_output[size] = keys[i];
    }
    ++values_output[size];
  }

  return size + 1;
}

size_t size_of_reduced_cpu (const std::vector<int> values) {
  size_t size = 0;
  for (size_t i = 0; i < values.size(); ++i) {
    if (values[i] % 2 != 0) {
      ++size;
    }
  }
  return size;
}

void copy_indices_cpu (std::vector<int> &output,
                       const std::vector<int> keys,
                       const std::vector<int> values,
                       size_t size) {
  size_t index = 0;
  std::vector<int> indices(size);
  for (size_t i = 0; i < size; ++i) {
    if (values[i] % 2 != 0) {
      output[index] = keys[i];
      ++index;
      indices[i] = 1;
    } else {
      indices[i] = 0;
    }
  }
}

std::vector<int> anf_product_reduce_cpu (std::vector<int> left_anf,
                                         std::vector<int> right_anf) {
  std::vector<int> output(left_anf.size() * right_anf.size());

  for (size_t i = 0; i < left_anf.size(); ++i) {
```

```
329       for (size_t j = 0; j < right_anf.size(); ++j) {
330         auto index = (i * right_anf.size()) + j;
331         output[index] = left_anf[i] | right_anf[j];
332       }
333     }
334
335     std::sort(output.begin(), output.end());
336
337     std::vector<int> keys(output.size());
338     std::vector<int> values(output.size());
339
340     size_t reduce_by_key_size = anf_reduce_by_key_cpu(output, keys, values);
341
342     size_t reduced_size = size_of_reduced_cpu(values);
343
344     std::vector<int> reduced(reduced_size);
345
346     copy_indices_cpu(reduced, keys, values, reduce_by_key_size);
347
348     return reduced;
349   }
350
351   int main(int argc, char** argv)
352   {
353     compute::device gpu = compute::system::default_device(); // gpu.name()
354     compute::context ctx(gpu);
355     compute::command_queue queue(ctx, gpu, CL_QUEUE_PROFILING_ENABLE);
356
357           std::string seq_opt = "--seq";
358           bool has_sequential_run = argc == 2 && seq_opt.compare(argv[1]) == 0;
359
360     auto tb1 =  std::chrono::steady_clock::now();
361     compute::program program;
362
363     try {
364       if (false && file_exists("/tmp/reduce.cl.bin")) {
365         program = compute::program::create_with_binary_file("/tmp/reduce.cl.bin",
366                                                             ctx);
367       } else {
368         program = compute::program::create_with_source_file("./reduce.cl", ctx);
369       }
370       program.build(); // attempt to compile to program
371       auto program_binary = program.binary();
372       std::cout << "Program binary size: " << program_binary.size() << std::endl;
373       writeFileBytes("/tmp/reduce.cl.bin", program_binary);
374     } catch(boost::compute::opencl_error &e) {
375       std::cout << program.build_log() << std::endl;
376       return -1;
377     }
```

Code

```
378
379   auto tb2 =  std::chrono::steady_clock::now();
380   auto timeBuildingKernels =
381     std::chrono::duration_cast<std::chrono::milliseconds>(tb2 - tb1).count();
382   std::cout << "Time spent building reduce.cl: "
383             << timeBuildingKernels << " ms" << std::endl;
384
385   prepareBoostComputeKernels(ctx, queue);
386
387   std::vector<std::vector<int>> anfs = parseProgramStdIn();
388
389   compute::vector<int> v_tmp(anfs[0].size(), ctx);
390   compute::copy(anfs[0].begin(), anfs[0].end(), v_tmp.begin(), queue);
391
392   int acc = 0;
393
394   std::vector<int> v_tmp_cpu = anfs[0];
395
396   std::vector<std::vector<int>> anfsTail(anfs.begin() + 1, anfs.end());
397   for (auto v = anfsTail.begin(); v != anfsTail.end(); ++v) {
398                 /* GPU run */
399     auto t1 = std::chrono::steady_clock::now();
400     compute::vector<int> vd(v->size(), ctx);
401     compute::copy(v->begin(), v->end(), vd.begin(), queue);
402     v_tmp = anf_product_reduce(v_tmp, vd, program, ctx, queue);
403     auto t2 = std::chrono::steady_clock::now();
404
405                 /* GPU copy */
406     auto t1_gpu_copy = std::chrono::steady_clock::now();
407     std::vector<int> gpu_result(v_tmp.size());
408     compute::copy(v_tmp.begin(), v_tmp.end(), gpu_result.begin(), queue);
409     auto t2_gpu_copy = std::chrono::steady_clock::now();
410
411     /* Sequential check */
412     auto t1_cpu = std::chrono::steady_clock::now();
413                 if (has_sequential_run) {
414                         v_tmp_cpu = anf_product_reduce_cpu(v_tmp_cpu, *v);
415                 }
416     auto t2_cpu = std::chrono::steady_clock::now();
417
418                 /* Log results */
419     std::cout << "acc "
420       << std::setw(3) << acc << ": "
421       << std::setw(7)
422       << std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count()
423       << " µs (gpu)"
424       << std::setw(7)
425       << std::chrono::duration_cast<std::chrono::microseconds>(t2_cpu - t1_cpu).count()
426       << " µs (cpu)"
```

```
427         << "\tsize: " << std::setw(6) << v_tmp.size() << " (gpu)"
428         << std::setw(6) << v_tmp_cpu.size() << " (cpu)"
429         << "\tcopy: " << std::setw(5)
430         << std::chrono::duration_cast<std::chrono::microseconds>(t2_gpu_copy - t1_gpu_copy
431         << " µs (gpu -> cpu)"
432         << std::endl;
433
434       // if (v_tmp_cpu != gpu_result) {
435       //  std::cout << "[ERROR] Vectors don't match" << std::endl;
436       //  printVector(gpu_result, "gpu: ", 10);
437       //  printVector(v_tmp_cpu, "cpu: ", 10);
438       //  std::cout << "--------" << std::endl;
439       //  return -1;
440       // }
441       acc++;
442     }
443
444     std::vector<int> result(v_tmp.size());
445     compute::copy(v_tmp.begin(), v_tmp.end(), result.begin(), queue);
446
447     printVector(result, "result (10):", 10);
448 }
```

A.1.3. Conversion to ANF (CPU)

```
1  {-# LANGUAGE DeriveAnyClass #-}
2  {-# LANGUAGE DeriveGeneric #-}
3  {-# LANGUAGE DeriveTraversable #-}
4  {-# LANGUAGE DerivingStrategies #-}
5  {-# LANGUAGE GeneralizedNewtypeDeriving #-}
6
7  module Formula.ANF where
8
9  import Control.DeepSeq (NFData)
10 import GHC.Generics (Generic)
11
12 import qualified Data.List as L
13
14 import qualified Formula.Prop as P -- ADT of a propositional formula
15
16 newtype ANF a = ANF {getANF :: XOR a}
17     deriving (Show, Eq, Ord, Functor, Foldable, Traversable, Generic)
18     deriving anyclass (NFData)
19
20 newtype XOR a = XOR {getXOr :: [AND a]} -- EmptyXOR
21     deriving (Show, Eq, Functor, Foldable, Traversable, Generic)
22     deriving anyclass (NFData)
23
24 newtype AND a = AND {getAnd :: [VAR a]}
```

```haskell
    deriving (Show, Eq, Functor, Foldable, Traversable, Generic)
    deriving anyclass (NFData)

data VAR a = ONE | VAR a
    deriving (Show, Eq, Functor, Foldable, Traversable, Generic)
    deriving anyclass (ToJSON, NFData)

instance Ord a => Ord (XOR a) where
    (XOR l) `compare` (XOR r)
        | length l < length r = LT
        | length l > length r = GT
        | length l == length r = L.sort l `compare` L.sort r

instance Ord a => Ord (AND a) where
    (AND l) `compare` (AND r)
        | length l < length r = LT
        | length l > length r = GT
        | length l == length r = L.sort l `compare` L.sort r

instance Ord a => Ord (VAR a) where
    ONE `compare` ONE = EQ
    ONE `compare` (VAR _) = LT
    (VAR _) `compare` ONE = GT
    (VAR a) `compare` (VAR b) = a `compare` b

(/+/) :: Eq a => XOR a -> XOR a -> XOR a
-- l /+/ EmptyXOR = l
-- EmptyXOR /+/ r = r
(XOR l) /+/ (XOR r) = XOR $ l ++ r

(/*/) :: Eq a => XOR a -> XOR a -> XOR a
-- l /*/ EmptyXOR = l
-- EmptyXOR /*/ r = r
(XOR l) /*/ r = foldr (/+/) (XOR []) [l' /./ r | l' <- l]

(/./) :: Eq a => AND a -> XOR a -> XOR a
-- a /./ EmptyXOR = XOR [ a /^/ a]
a /./ (XOR b) = XOR [a /^/ b' | b' <- b]

(/^/) :: Eq a => Eq a => AND a -> AND a -> AND a
(AND l) /^/ (AND r) = filterOne . AND $ l ++ r

one :: XOR a
one = XOR [AND [ONE]]

var :: a -> XOR a
var a = XOR [AND [VAR a]]

fromProp :: Ord a => P.Prop a -> ANF a
```

Code

```haskell
fromProp = ANF . canonicalXOR . fromProp'

fromProp' :: Eq a => P.Prop a -> XOR a
fromProp' (P.XOr a b) = fromProp' a /+/ fromProp' b
fromProp' (P.And a b) = fromProp' a /*/ fromProp' b
-- a || b = ab + a + b
fromProp' (P.Or a b) = (fromProp' a /*/ fromProp' b) /+/ fromProp' a /+/ fromProp' b
-- a => b = ab + a + 1
fromProp' (P.Imp a b) = (fromProp' a /*/ fromProp' b) /+/ fromProp' a /+/ one
-- a => b = ab + a + 1
fromProp' (P.Iff a b) = fromProp' a /+/ fromProp' b /+/ one
-- not a = a + 1
fromProp' (P.Not a) = fromProp' a /+/ one
fromProp' (P.Var v) = var v
fromProp' P.T = one
fromProp' P.F = XOR []

-- | Reduce to canonical form
canonical :: Ord a => ANF a -> ANF a
canonical = ANF . canonicalXOR . getANF

canonicalXOR :: Ord a => XOR a -> XOR a
canonicalXOR = XOR . filterByPair . L.sort . L.filter (/= AND []) . fmap canonicalAND .

canonicalAND :: (Ord a) => AND a -> AND a
canonicalAND = AND . L.nub . L.sort . getAnd . filterOne

-- | 1 x a = a
filterOne :: Eq a => AND a -> AND a
filterOne (AND a)
    | L.all (== ONE) a = AND [ONE]
    | ONE `L.elem` a = AND $ L.filter (/= ONE) a
    | otherwise = AND a

-- | a + a + b = b
filterByPair :: Eq a => [a] -> [a]
filterByPair [] = []
filterByPair [x] = [x]
filterByPair (x : y : ys)
    | x == y = filterByPair ys
    | otherwise = x : filterByPair (y : ys)
```