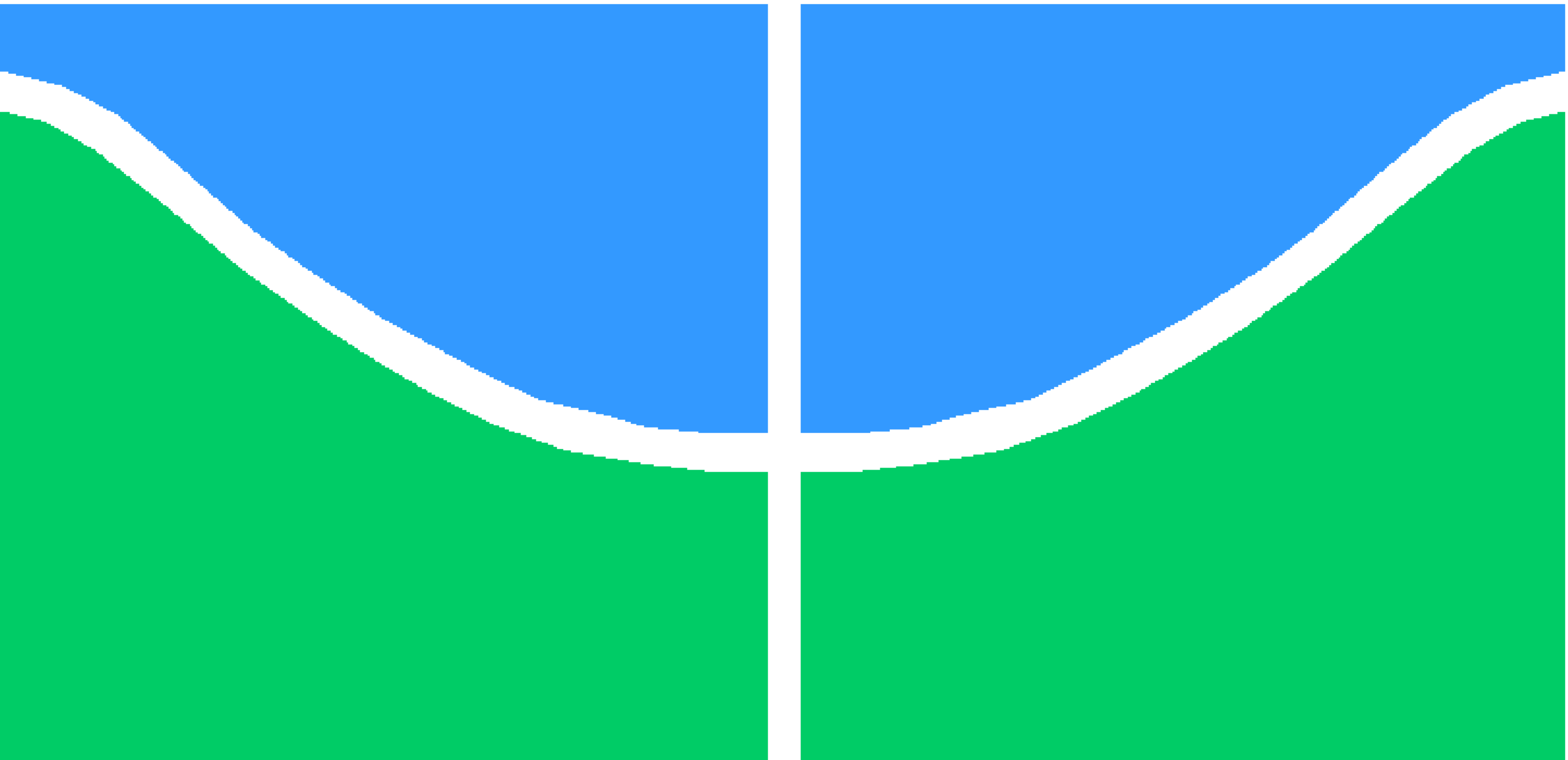

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Uma Implementação Paralela em GPU do Algoritmo Walksat

Autor: Luís Henrique Pereira Taira

Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF

2022

Luís Henrique Pereira Taira

Uma Implementação Paralela em GPU do Algoritmo Walksat

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF

2022

Resumo

Resolverodores SAT são os programas de computador que resolvem o problema da Satisfatibilidade, que, por ser um dos problemas NP-Completos e por suas aplicações, é um importante problema para a Ciência da Computação. Por isso, várias áreas do conhecimento são impactadas quando melhor desempenho é alcanlçado por Resolvedores SAT.

O uso de processamento paralelo é uma forma comum de se realizar grandes quantidades de computação no menor tempo possível. Portanto é interessante utilizar paralelização em Resolvedores SAT para obter o melhor desempenho possível.

Este trabalho teve como objetivo investigar o potencial de melhora do desempenho de um Resolvedor SAT, baseado no algoritmo Walksat, mediante a aplicação de processamento paralelo. Para isso foram desenvolvidos dois Resolvedores SAT baseados no Walksat, um serial e um paralelo que usa aceleração de GPU (Graphical Processing Unit) para que pudesse ser feita uma análise comparativa de seus desempenhos.

Os resultados obtidos por meio de testes demostraram que a implementação paralela teve desempenho significativamente pior do que as implementações seriais. Os resultados dos testes mostraram também que a implementação paralela obtida não utiliza todo o potencial de processamento paralelo de uma GPU. Portanto teoriza-se que uma implementação melhor otimizada do Walksat para GPU possa alcançar desempenho melhor.

Palavras-chaves: satisfatibilidade booleana, lógica proposicional, paralelização, Walksat, GPU, CUDA.

Abstract

The objective of this research was to investigate the performance gains of a SAT Solver based on the Walksat algorithm when parallel processing is used. For this purpose, two Walksat based SAT Solvers were developed, one based on serial processing and the other on parallel processing using GPU (Graphical Processing Unit) acceleration, so that a comparative analysis of their performance could be made.

SAT Solvers are computer programs that solve the Satisfiability problem, which is an important topic for Computer Science because of its applications and for being an NP-Complete problem; therefore, many areas of knowledge are impacted when SAT Solvers achieve better performance.

A common way to execute large amounts of computation in the shortest amount of time possible is to use parallel processing. Hence, it is of interest, to use parallelization to achieve the best possible performance in SAT Solvers.

The results, that were obtained via tests, demonstrated that the parallel implementation had significantly worse performance than the serial implementations. The results also showed that the parallel solution does not utilize all of the parallel processing potential of a GPU and thus it is theorized that a better optimized implementation for GPU of the Walksat algorithm could accomplish better performance.

Key-words: boolean satisfiability, propositional logic, paralellization, Walksat, GPU, CUDA.

Lista de ilustrações

Figura 1 – Quadro de Blocks 19
Figura 2 – Exemplo NVVP 20
Figura 3 – NVVP implementação otimizada em CUDA 26
Figura 4 – NVVP primeira implementação paralela em CUDA 27
Figura 5 – Desempenho dos algoritmos por cláusulas (escala logarítmica) 32
Figura 6 – Desempenho dos algoritmos por variáveis 33
Figura 7 – Desempenho dos algoritmos por razão (escala logarítmica) 34

Lista de tabelas

Tabela 1 – Desempenho dos algoritmos por cláusulas 32
Tabela 2 – Desempenho dos algoritmos por variáveis 33
Tabela 3 – Desempenho dos algoritmos por razão 34

Lista de abreviaturas e siglas

SAT	Satisfatibilidade Booleana
NP	Nondeterministic Polynomial
CPU	Central processing unit
GPU	Graphics processing unit
API	Application Programming Interface
MIMD	Multiple Instruction Multiple Data
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
MT/s	Megatransfers per second

Sumário

1	**INTRODUÇÃO**	**9**
1.1	**Problemas de satisfatibilidade**	**9**
1.2	**Algoritmos Incompletos**	**9**
1.2.1	Algoritmos de busca local	10
1.3	**Paralelização de algoritmos SAT**	**10**
1.4	**CPUs e GPUs**	**11**
1.5	**Objetivos**	**12**
1.5.1	Objetivo Central	12
1.5.2	Objetivos Específicos	12
2	**REFERENCIAL TEÓRICO**	**13**
2.1	**Introdução a Lógica Booleana**	**13**
2.2	**Satisfatibilidade**	**13**
2.2.1	SAT é NP-Completo	14
2.3	**Algoritmos incompletos**	**14**
2.3.1	Busca local estocástica	15
2.3.2	Operações comuns	15
2.3.2.1	*Flip*	15
2.3.2.2	Cálculo do *BreakCount*	15
2.3.2.3	Cálculo do *MakeCount*	15
2.3.2.4	GSAT	15
2.4	**Programação para GPU**	**16**
2.4.1	Paradigmas	16
2.4.2	Tecnologia	17
2.4.2.1	OpenCL	17
2.4.2.2	OpenMP	18
2.4.2.3	NVC / NVFORTRAN	18
2.4.2.4	CUDA	18
2.4.2.4.1	Desenvolvimento com CUDA	19
2.4.2.4.2	NVIDIA Visual Profiler	20
2.4.2.5	Uso de Memória	20
3	**WALKSAT**	**22**
4	**IMPLEMENTAÇÃO DO WALKSAT EM GPU**	**24**
4.1	**Metodologia**	**24**

4.2 **Metodologia de paralelização do Walksat** **25**
4.3 **Implementação paralela** **25**
4.3.1 Versão Inicial 26
4.3.2 Otimizações 26
4.3.3 Algoritmo Paralelo 27
4.4 **Resultados** **29**
4.4.1 Desempenho em relação ao número de cláusulas e variáveis 31
4.4.2 Desempenho em relação à razão de variáveis por cláusulas 32
4.4.3 Análise 34

5 **CONCLUSÃO** **36**

REFERÊNCIAS **37**

1 Introdução

Lógica proposicional é uma área da Matemática iniciada por Aristóteles e Teofrasto no século IV A.E.C como forma de interpretar o pensamento humano, representando-o em declarações que são sempre ou verdadeiras ou falsas, mas nunca ao mesmo tempo, podendo-se, nelas, aplicar operadores lógicos **E** ($\wedge$) e **OU** ($\vee$). Já no século XIX, George Boole criou o modelo que usamos hoje para simbolizar declarações e operações em expressões algébricas que passaram a ser conhecidas como expressões booleanas. A história da Lógica Proposicopnal é melhor detalhada por Franco e Martin (2009).

1.1 Problemas de satisfatibilidade

Segundo Lee, Roychowdhury e Seshia (2011), para uma dada função Booleana $F(x_1, x_2, x_3, \ldots, x_n)$, que recebe n variáveis x_i de entrada, é um problema de Satisfatibilidade (SAT) determinar se, para alguma combinação de valores de $x_1 - x_n$, é possível que F tenha valor verdadeiro. E, se sim, retornar os valores de $x_1 - x_n$. Hoje em dia, segundo Sinz (2005), Franco e Martin (2009) e como demonstrado por Marques-Silva (2008), o problema da Satisfatibilidade é relevante para várias áreas, especialmente na Ciência da Computação, no que se refere a, *design* de circuitos, inteligência artificial, verificação de software entre outros.

Para essas aplicações, se tornou importante resolver a Satisfatibilidade de funções da forma mais rápida e eficiente possível. Por isso é interessante utilizar paralelização para aumentar o desempenho de resolvedores.

Algoritmos SAT são divididos em dois grupos, completos e incompletos. Este trabalho foca em algoritmos incompletos.

1.2 Algoritmos Incompletos

Algoritmos incompletos são assim chamados pois não dão garantia que vão encontrar a solução de uma função ou declarar que ela é insatisfazível.

De acordo com Kautz, Sabharwal e Selman (2008), algoritmos incompletos têm a tendência de, quase sempre, conseguirem a solução de funções satisfazíveis mas falham em declarar uma conclusão para algumas funções insatisfazíveis. Pode-se afirmar que algoritmos assim tendem à satisfação.

Outrossim, é possível, em teoria, que um algoritmo incompleto tenda à insatisfação, sempre conseguindo provar quando uma fórmula é insatisfazível, mas falhando em resolver

certos casos satisfazíveis.

1.2.1 Algoritmos de busca local

Algoritmos incompletos são geralmente baseados em busca local estocástica, os melhores incorporam ruído na forma de atribuições aleatórias para variáveis e escolha aleatória de cláusulas. Por isso a busca local é uma das principais vertentes dos algoritmos SAT.

O algoritmo GSAT é um dos principais algoritmos de busca local. É importante, principalmente, porque forneceu uma base para outros algoritmos, considerados mais eficientes, como o GWSAT e o Walksat, explicado no Capítulo 3.

Kautz, Sabharwal e Selman (2008) explicam que o procedimento do GSAT começa pela atribuição de valores aleatórios às variáveis e então faz uma troca (*flip*) ambiciosa do valor da variável que resultará na maior diminuição do número de cláusulas não resolvidas. Essas trocas são realizadas repetidamente até que seja encontrada uma solução para a função ou que seja atingido um número máximo de trocas pré-determinado. O processo é repetido desde a atribuição aleatória das variáveis até que se encontre uma solução ou que seja atingido um número máximo de tentativas.

Para esse estudo, optou-se pelo uso do Walksat, tendo em vista ser um algoritmo bem documentado, baseado em métodos de algoritmos clássicos, como a Busca Local e a Subida de Encosta.

1.3 Paralelização de algoritmos SAT

Algoritmos SAT podem ser desenvolvidos para usar paralelização, uma vez que podem ter partes de seu cálculo escrito para que mais de uma operação seja feita ao mesmo tempo, diminuindo efetivamente o tempo levado para que o cálculo termine. Existem muitas implementações paralelas de Resolvedores SAT, como por exemplo o resolvedor feito por McDonald (2009).

A efetividade de algoritmos SAT paralelos depende de diversas variáveis como: o número de atribuições de verdade, de variáveis, de cláusulas de uma função, do algoritmo que esta sendo usado etc. A diferença de tempo entre um algoritmo paralelo e um puramente serial pode chegar perto de ser proporcional ao número máximo de instâncias que podem ser criadas ou não ter nenhuma diferença, existindo ainda a possibilidade de que, em casos menos favoráveis, ao paralelizar a solução, ocorra aumento do tempo de execução, devido ao uso ineficiente da capacidade de processamento paralelo e do custo computacional adicional, que pode ser necessário para a execução de uma aplicação paralela, como cópias de dados entre memória principal e do acelerador ou por causa de

overhead causado pela API escolhida.

1.4 CPUs e GPUs

A maioria dos programas de computador e também dos resolvedores SAT são escritos para que seu processamento ocorra nas Unidades de Processamento Central (Central Processing Unit - CPU), ou processadores, dos computadores. Portanto é nos processadores onde ocorre a maior parte do processamento feito por um computador.

Os processadores geralmente possuem mais de um núcleo, o que lhes permite realizar mais de uma operação a cada ciclo de processamento. Isso é o que se chama de paralelização

As Unidades de Processamento Gráfico (Graphical Processing Units - GPU), ou placas de vídeo, são aceleradores que podem ser adicionados a um computador. Foram criadas para acelerar o processamento de aplicações gráficas. Porém, pela sua alta capacidade de processamento paralelo, passaram a ser usadas para outras aplicações computacionais, com capacidade de se beneficiarem com a paralelização, por isso existem hoje GPUs feitas para serem usadas nessas aplicações e não só em aplicações gráficas, como as da linha QUADRO e TESLA da NVIDIA e FirePro da AMD.

A capacidade de processamento serial de GPUs é bem menor do que a de CPUs. Para computação puramente serial não vale a pena usar GPUs, assim como para o uso em aplicações paralelas que não são capazes de usar a paralelização de forma eficiente.

Contudo, quando se trata de computação paralela, pode ocorrer grande ganho de desempenho de uma aplicação paralela, quando se comparada à mesma aplicação rodando serialmente em CPU.

No estudo realizado por Liu, Hu e Li (2018), foi encontrado o potencial de aumento de desempenho em mais de 100x quando é realizada paralelização em GPU de certas atividades e na ordem de, no máximo, 10x quando realizada paralelização em CPU. Este nível de melhora ao se paralelizar em GPU é derivado de casos ideais, por isso o que se espera de uma paralelização em GPU de aplicações reais é de no máximo 30x.

Apesar de programas não precisarem sempre ser escritos especificamente para rodarem nas GPUs, a maioria das tecnologias têm esse requisito.

Existem Resolvedores SAT que utilizam processamento em GPUs, como o feito por Osama, Wijs e Biere (2021) e o Gpu4sat de Deleau, Jaillet e Krajecki (2008).

1.5 Objetivos

1.5.1 Objetivo Central

O objetivo deste trabalho é investigar o potencial de melhora de um Resolvedor SAT baseado no algoritmo Walksat ao se aplicar processamento paralelo.

1.5.2 Objetivos Específicos

- Desenvolver um Resolvedor SAT, baseado no Walksat, que use processamento em GPU;
- desenvolver um Resolvedor SAT serial, baseado no Walksat, que tenha implementação parecida ao resolvedor paralelo;
- compreender os fatores que afetam a diferença entre o desempenho de resolvedores baseados no Walksat seriais e paralelos.

Este trabalho está dividido em cinco capítulos. No Capítulo 2 está o referencial teórico do trabalho, no Capítulo 3, uma explicação detalhada sobre o Walksat, no Capítulo 4 é descrito o trabalho realizado os resultados são analisados e no Capítulo 5 é apresentada uma conclusão.

2 Referencial Teórico

2.1 Introdução a Lógica Booleana

De acordo com Klement (2022) e Franco e Martin (2009), o modelo de lógica proposicional inventado por Aristóteles no Século século IV A.E.C, é focado no que se chama de declarações, como, por exemplo, "está chovendo". Esta declaração pode ser verdadeira ou falsa. Também foi proposto por Aristóteles ou por Teofrasto que operadores lógicos como **OU**, **E** ou **não**, podem ser usados em uma ou mais declarações. Assim, duas declarações unidas por um operador, como por exemplo, "Está chovendo **E** está de noite", formam uma nova declaração, que tem um sentido próprio.

Da mesma forma, ao se negar uma declaração, por meio do uso do operador **não**, forma-se uma nova declaração, com sentido oposto a original. Isto é, sempre que a declaração "está chovendo" for falsa, a afirmação "**não** está chovendo" será verdadeira.

No próprio modelo lógico de Aristóteles, pela lei do terceiro excluído e a lei da contradição, uma declaração sempre tem valor verdadeiro ou falso e nunca é verdadeira e falsa ao mesmo tempo.

A forma com que a lógica proposicional se manifesta hoje em dia, se dá, principalmente, por meio da abstração algébrica que se usa para representar expressões, que foi criada por George Boole no Século XIX. Por esse motivo, o tratamento de expressões lógicas matematicamente ficou conhecido como Lógica Booleana.

O que na Lógica Aristotélica se chama de declarações ou afirmações, na Lógica Booleana são variáveis que carregam um valor que pode ser "verdadeiro" ou "falso".

Denomina-se cláusula um conjunto de variáveis unidas por operadores **OU**, representados pelo símbolo: $\vee$.

Uma função Booleana, para fins deste trabalho, é um conjunto de múltiplas cláusulas unidas por operadores **E**, representados pelo símbolo: $\wedge$. O operador **não** é usado e é representado pelo símbolo: $\neg$.

A função $F_1 = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$ é um exemplo de função Booleana, com duas cláusulas e três variáveis.

2.2 Satisfatibilidade

Na definição de Larrosa, Lynce e Marques-Silva (2010), o Problema da Satisfatibilidade Booleana consiste em decidir se uma combinação de valores torna satisfeita

uma fórmula booleana quando atribuídos a suas variáveis. Funções pequenas, com poucas cláusulas, podem ter sua Satisfatibilidade determinada facilmente por uma pessoa, mas para funções com dezenas de cláusulas ou mais, faz se necessário utilizar computação, especialmente quando se quer testar vários valores diferentes para cada variável em uma função.

Como explicado por Marques-Silva (2008), o problema da satisfatibilidade é importante para várias áreas, especialmente da Ciência da Computação, como: *design* de circuitos eletrônicos, testagem de modelos, testagem de software e circuitos, Inteligência Artificial e halotipagem em Bioinformática.

2.2.1 SAT é NP-Completo

De acordo com a enciclopédia Britannica (2022), são problemas da classe NP (*Non-deterministic Polynomial*), aqueles que podem ter uma solução adivinhada e verificada em tempo polimonial. Algoritmos que executam em tempo polinomial são considerados eficientes, enquanto os que executam em tempo exponencial são considerados não eficientes.

Um problema que faz parte do conjunto NP e para o qual todos os problemas NP podem ser convertidos em tempo polinomial é chamado de um problema NP-Completo.

O problema da Satisfatibilidade foi o primeiro a ser provado NP-Completo pelo teorema de Cook-Levinson (Escoffier e Paschos, 2006). Isso significa que a existência de uma solução eficiente para o problema da Satisfatibilidade implicaria em uma solução eficiente para todos os outros problemas NP-Completos. Não existe, porém, uma solução polinomial para um problema NP-Completo ainda.

2.3 Algoritmos incompletos

Resolvedores SAT recebem, como entrada, funções Booleanas, com cada uma de suas cláusulas, estas sem limitação de tamanho, e são capazes de testar várias atribuições de valores para variáveis. O objetivo desses algoritmos é provar que uma função é satisfazível ou não e, se sim, dizer quais valores de variáveis a tornam verdadeira.

De acordo com Kautz, Sabharwal e Selman (2008), um algoritmo é chamado de completo quando é capaz de sempre determinar que uma função é ou não satisfazível. Algoritmos incompletos não dão garantia que vão eventualmente encontrar a Satisfatibilidade de uma função.

No *Handbook of Satisfiability*, Kautz, Sabharwal e Selman (2008) dizem que a algoritmos incompletos geralmente são tendenciosos para uma direção e têm incerteza em um lado, porque geralmente são competentes em encontrar soluções para funções

satisfazíveis quase todas as vezes mas têm mais dificuldade de apontar quando uma função é insatisfazível, obtendo falha em uma parte maior das vezes.

2.3.1 Busca local estocástica

Algoritmos incompletos geralmente se baseiam no mecanismo de começar a busca em algum ponto da fórmula e se movimentar localmente usando informações contidas somente na vizinhança do último ponto. Esse método se chama *Busca local estocástica* e algoritmos baseados nele tendem a ter desempenho significativamente melhor do que algoritmos completos baseados em DPLL (Davis-Putnam-Logemann-Loveland) (Kautz, Sabharwal e Selman, 2008).

2.3.2 Operações comuns

Durante a execução de um algoritmo de Satisfatibilidade, algumas operações compõem grande parte do raciocínio de alguns algoritmos e geralmente são realizadas mais do que uma vez.

2.3.2.1 *Flip*

Quando um resolvedor está realizando um cálculo, são atribuídos valores a cada uma das variáveis contidas nas funções. Para encontrar um valor que resolva a função, parte do raciocínio geralmente inclui realizar inversões no valor de variáveis. Quando o valor e alguma variável é invertido, se diz que foi realizado um *flip* nesta variável.

2.3.2.2 Cálculo do *BreakCount*

Quando cada variável de uma função tem um valor, um certo número de cláusulas se encontram satisfeitas e um outro número se encontra não satisfeitas. Denomina-se o *BreakCount* de uma variável, o número de cláusulas que passariam de satisfeitas para insatisfeitas caso seja feito um *flip* nessa variável.

2.3.2.3 Cálculo do *MakeCount*

Similarmente ao *BreakCount*, denomina-se o*MakeCount* de uma variável o número de cláusulas que passariam de insatisfeitas para satisfeitas caso seja feito um *flip* nessa variável.

2.3.2.4 GSAT

O GSAT, descrito no Algoritmo 1 é uma implementação ambiciosa e aleatorizada do mecanismo de busca local estocástica. O procedimento do GSAT começa com uma atribuição de valores gerada aleatoriamente para todas as variáveis e faz um *flip* na variável

com o menor *BreakCount*. Esses *flips* são feitos até que uma atribuição satisfatória seja encontrada ou um número máximo de *flips* seja atingido. O processo é repetido até que seja encontrada uma solução para a função ou seja atingido o número máximo de tentativas.

Algoritmo 1: GSAT(F)

```
Input: A CNF formula F
Data: Integers MAX-FLIPS, MAX-TRIES
Output: A satisfying assignment for F, or FAIL
1 for i ← 1 to MAX − TRIES do
2   σ ← a randomly generated truth assignment for F;
3   for j ← 1 to MAX − FLIPS do
4     if σsatisfies F then return σ;                     // success
5     v ← a variable flipping which results in the greatest decrease
6       (possibly negative) in the number of unsatisfied clauses
7     Flip v in σ;
8   return FAIL;                      // no satisfying assignment found
9
```

Fonte: (KAUTZ; SABHARWAL; SELMAN, 2008)

Um importante algoritmo que se baseia no GSAT é o Walksat, melhor explicado no Capítulo 3, que introduz duas importantes mecânicas:

- Depois que uma cláusula insatisfeita C é selecionada aleatoriamente, caso exista em C uma variável x com $BreakCount = 0$ é feito um *flip* em x;
- caso não exista essa variável, existe probabilidade p de que seja feito um *flip* na variável em C com o menor *BreakCount*, e nos casos restantes o *flip* é feito em uma variável escolhida aleatoriamente em C.

2.4 Programação para GPU

GPUs são aceleradores de processamento, os quais são equipamentos de hardware que um computador pode usar para realizar parte do processamento necessário para uma aplicação. Assim sendo, não podem ser usadas como substitutos para CPUs mas os dois devem ser usados em conjunto.

2.4.1 Paradigmas

De acordo com Mattson, Sanders e Massingill (2005), a Taxonomia de Flynn, proposta por Michael J. Flynn em 1966, é o padrão usado para se classificar computadores paralelos e os divide nas seguintes classificações:

- SISD: Single Instruction, Single Data.
- SIMD: Single Instruction, Multiple Data.
- MISD: Multiple Instruction, Single Data.
- MIMD: Multiple Instruction, Multiple Data.

A arquitetura, na qual o modelo de execução de GPUs modernas da NVIDIA se baseia, é a SIMD. Porém, na prática, o modelo de execução usa a arquitetura SIMT (Single Instruction Multiple Thread), que existe como uma extensão da Taxonomia de Flynn sendo baseada na SIMD, e é definida no *CUDA C++ Programming Guide* (NVIDIA, 2022a).

A SIMT consiste na separação em threads de todas as instruções mandadas ao multiprocessador, cada uma podendo ter sua própria instrução diferente das outras e seu próprio conjunto de dados. SIMT é diferente de MIMD porque não permite que as instruções executadas em paralelo pelo multiprocessador sejam diferentes entre si sem que haja perda de desempenho.

As threads que são enviadas ao multiprocessador são separadas em trinta e duas, formando conjuntos, ou *warps*, que são enfileirados para execução em paralelo. Apesar de cada thread poder ter uma instrução diferente, o melhor desempenho é obtido quando todas as threads de um conjunto têm a mesma instrução. Portanto, o conceito de se paralelizar código sem ou com poucas ramificações, para obter bom desempenho, ainda se aplica.

Devido ao uso de conjuntos, de acordo com a Seção 4.1 do *CUDA C++ Programming Guide* (NVIDIA, 2022a), é possível hoje em dia a paralelização de código com poucas ramificações sem que haja perda de desempenho em casos que seja possível montar conjuntos com a mesma instrução.

A programação paralela feita para GPU é diferente da feita para CPU, porque CPUs usam o paradigma MIMD (RASTERGRID, 2022)

2.4.2 Tecnologia

Existem diversas formas de se fazer programação paralela. Porém nem todas são capazes de alocar processamento para GPUs. Algumas tecnologias são consideradas as principais para se trabalhar com GPU.

2.4.2.1 OpenCL

OpenCL (Open Computing Language) é uma linguagem de programação e um *framework* de computação totalmente *open source* e é compatível com GPUs de todos

os principais fabricantes do mercado e ainda com vários outros tipos de aceleradores computacionais como FPGAs e processadores Tensor.

OpenCL é uma linguagem diferente, porém ainda remanescente de C++ e que requer um nível mais elevado de aprendizado do que CUDA e apresenta produtividade menor, o que é demonstrado no estudo de Memeti et al. (2017).

OpenCL também é muito difundida no mercado e é usada em diversas aplicações de computação até em super computadores.

2.4.2.2 OpenMP

OpenMP é uma API composta de diversas bibliotecas de código disponíveis para serem usadas diretamente em ambientes C, C++ ou Fortran que tem o intuito específico de facilitar o desenvolvimento de aplicações computacionais paralelas (OPENMP, 2018).

O OpenMP é, a muitos anos, uma tecnologia bastante utilizada, porém focada em programação paralela em CPUs, sendo também possível usar o OpenMP com aceleração de GPU, mas com desempenho bem menor do que OpenCL e CUDA, como demonstrado por Memeti et al. (2017).

2.4.2.3 NVC / NVFORTRAN

O NVC e o NVFORTRAN são compiladores desenvolvidos pela NVIDIA, criados para facilitar fortemente o desenvolvimento de aplicações paralelas, possibilitando a paralelização de programas inicialmente escritos para serem seriais com muito pouco esforço.

O desenvolvimento de aplicações paralelas com NVC é feito em C, C++ ou Fortran padrão, pois contém implementações paralelas de funções padrões das linguagens e, por isso, é sem dúvida a opção com maior facilidade de implementação (NVIDIA, 2022b).

Apesar de ser compatível com CPUs de vários fabricantes, o NVC e NVFORTRAN são compatíveis com GPUs somente da NVIDIA, assim como CUDA.

2.4.2.4 CUDA

CUDA é uma plataforma de computação paralela de software e hardware desenvolvida pela NVIDIA para ser uma ferramenta que desenvolvedores possam usar para acelerar a computação de aplicações.

CUDA é uma tecnologia proprietária e que só pode ser usada em GPUs fabricadas pela própria NVIDIA, mesmo assim é extremamente difundida no mercado.

CUDA foi escolhida para o desenvolvimento deste trabalho devido à grande quantidade de documentação de boa qualidade disponível e de discussões em fóruns que pos-

sibilitam a rápida resolução de problemas e por possuir o melhor desempenho dentre as alternativas (Memeti et al., 2017).

2.4.2.4.1 Desenvolvimento com CUDA

Diferentemente de OpenMP ou OpenCL, CUDA não é somente uma API que pode ser importada por meio de uma biblioteca, mas, sim, um ambiente que usa a própria linguagem chamada de CUDA C++, definida no *CUDA C++ Programming Guide* (NVIDIA, 2022a), fortemente baseada no C++ padrão, com funcionalidades e funções que permitem a integração com aceleradores gráficos.

Por ter seu próprio ambiente, programas em CUDA são compilados usando o próprio compilador da NVIDIA, o NVCC, que também é capaz de compilar código C e C++.

A alocação de processamento para GPUs em CUDA é feita por meio da execução de *kernels*, resumidamente, funções cujo código é executado somente nos aceleradores. São chamados de *kernels* e não de funções, porque sua execução ocorre de forma diferente.

Quando chamado no código, os *kernels* são enfileirados para execução em GPU já com o número predefinido de vezes que o códidigo será executado. Antes da execução de um *kernel*, precisa ser definido o número de *threads*, que é o número de instâncias paralelas necessárias, o número de blocos de *threads* e o número de *threads* por bloco, que se organizam conforme a Figura 1.

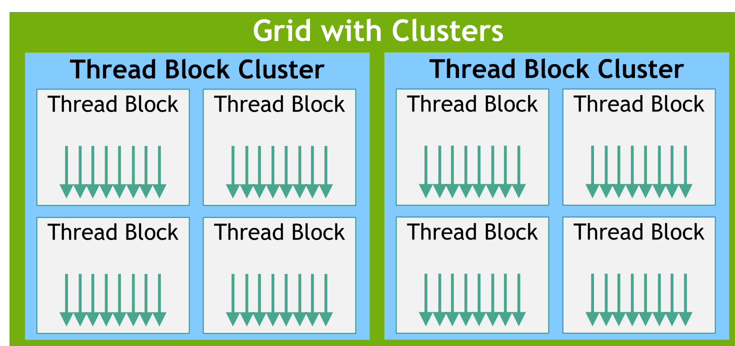

Figura 1 – Quadro de Blocks

Fonte: NVIDIA, (2022a)

O número de blocos e de threads por bloco precisam ser definidos e passados ao *kernel* em sua chamada porque são executados ao mesmo tempo vários blocos de no máximo 1024 *threads*. Cada *thread* tem seu próprio endereço em uma vetor tridimensional, o que facilita a leitura paralela de um conjunto de dados de até três dimensões.

2.4.2.4.2 NVIDIA Visual Profiler

O *NVIDIA Visual Profiler* (NVVP) é uma das ferramentas disponibilizadas pela NVIDIA como parte do *CUDA Toolkit* para desenvolvimento de aplicações CUDA e realiza, de forma automatizada, análises sobre o desempenho de um programa CUDA.

Mediante a escolha de um programa CUDA e sua execução, o NVVP provém importantes informações. Principalmente uma separação por tempo de execução sobre que tarefa o programa estava executando em cada momento. Um exemplo de informações providas pelo NVVP está na Figura 2.

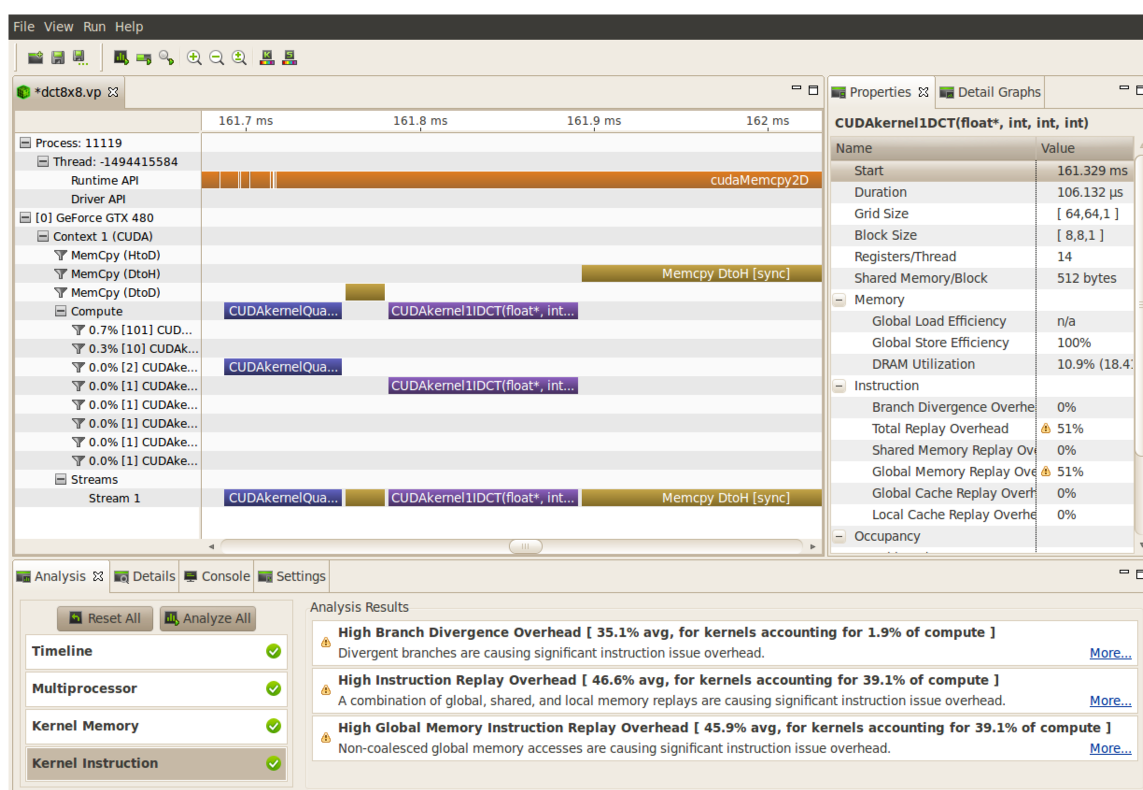

Figura 2 – Exemplo NVVP

Fonte: NVIDIA, (2022c)

Outras importantes informações são providas na seção *Analysis Results* da Figura 2, onde o NVVP apresenta os pontos que considera mais importantes sobre o desempenho obtido pela aplicação testada.

2.4.2.5 Uso de Memória

Uma importante consideração ao se planejar e programar algoritmos que usem aceleração em GPU é que todos os dados utilizados no código executado em GPU precisam estar presentes em sua memória no momento de seu uso. A transferência de dados entre a memória principal do computador e a da GPU pode ocorrer quando explicitamente

solicitada pelo programador ou de forma implícita e gerenciada pela implementação da API.

É necessário levar em conta transferências de dados ao realizar otimizações em programas que usam GPUs porque essas transferências ocorrem por meio do barramento PCIe que é mais devagar do que o barramento na qual é ligada a memória principal e muito mais lento do que a memória cache do CPU, que é a única usada quando o conjunto de dados é pequeno o bastante para caber nela.

A forma tradicional de se realizar transferência de memória entre GPU e memória principal é solicitando explicitamente. Em CUDA isso se manifesta na forma da função `cudaMemCpy` (NVIDIA, 2022a) e suas variantes. Nesse caso, é responsabilidade do programador decidir quando as cópias de dados ocorrem e, mais importante, decidir quando e em que ordem as operações devem ser feitas em CPU ou em GPU, levando em consideração o tempo de processamento que será necessário para transitar os dados para que seja possível realizar cada operação.

Outro mecanismo disponível é o de memória compartilhada (HARRIS, 2013) (INTEL, 2014). O programador pode optar por ter variáveis que estão sempre armazenadas na memória principal ou da GPU, mas que sejam acessíveis por qualquer uma das partes. Isso implica que, por exemplo, se uma variável está armazenada na memória principal, mas acessível pela GPU, quando essa variável é referenciada pela CPU, ocorre o uso de memória como se fosse uma variável ordinária e quando referenciada pela GPU, ocorre, implicitamente, a cópia dessa variável para a GPU no momento da referência. Se o conteúdo for modificado em GPU, o conteúdo é copiado para a memória principal. Também é possível que variáveis armazenadas na memória da GPU sejam acessíveis pelo CPU. Nesse caso, o mesmo processo ocorre invertido.

A terceira forma comumente usada é o conceito de memória unificada entre GPU e memória principal (HARRIS, 2013) (INTEL, 2014). Quando memória unificada é usada, o conteúdo de uma dada variável está presente, tanto na memória principal quanto na de GPU. Desta forma, quando uma das partes solicita acesso ao conteúdo da variável, não é necessária a cópia desse conteúdo antes que a operação seja realizada e, quando seu conteúdo é modificado, fica como responsabilidade da API decidir quando é feita a transferência de dados para outra parte e quando essa transferência pode ser feita de forma paralela, ao mesmo tempo que outras operações são realizadas.

Cada maneira de se utilizar memória têm seus benefícios e cabe ao programador escolher a forma mais adequada e escrever código que manipule memória de forma eficaz independentemente da forma escolhida.

3 Walksat

Walksat não dá certeza de encontrar a satisfatibilidade de todas as funções, porque é um algoritmo incompleto. Porém é hoje uma relevante técnica para resolver o problema da Satisfatibilidade.

A base para o funcionamento do Walksat é o princípio de busca local estocástica ambiciosa e o algoritmo GSAT, explicado na seção 2.3.2.4. Porém os mecanismos, desenbolvidos em 2004 por Kautz, Selman e Mcallester, o tornaram muito mais rápido do que seus antecessores.

Algoritmo 2: Walksat(F)

```
    Input: A CNF formula F
    Data: Integers MAX-FLIPS, MAX-TRIES; noise parameter p ∈ [0,1]
    Output: A satisfying assignment for F, or FAIL
 1  for i ← to MAX − TRIES do
 2  |  σ ← a randomly generated truth assignment for F;
 3  |  for j ← 1 to MAX − FLIPS do
 4  |  |  if σ satisfies F then return σ;                 // success
 5  |  |  C ← an unsatisfied clause of F chosen at random;
 6  |  |  if ∃ variable x ∈ C with break − count = 0 then
 7  |  |  |  v ← x;                                         // freebie move
 8  |  |  else
 9  |  |  |  With probability p :                          // random walk move
10  |  |  |     v ← a variable in C chosen at random;
11  |  |  |  With probability 1 − p :                      // greedy move
12  |  |  |     v ← a variable in C with the smallest break − count;
13  |  |  Flip v in σ;
14  |  return FAIL;                       // no satisfying assignment found
15  |
```

Fonte: (KAUTZ; SABHARWAL; SELMAN, 2008)

A implementação do Walksat representado no Algoritmo 2 recebe como parâmetros os valores:

- F: A fórmula a ser avaliada;
- MAX-FLIPS: máximo de iterações que podem ser realizados em uma tentativa;
- MAX-TRIES: máximo de tentativas que podem ser realizadas;

- p: ruído.

Por ser um SAT solver comum, esta implementação olha para uma função e tenta determinar uma atribuição de valores para suas variáveis que a torne verdadeira. Se encontrada, essa atribuição é retornada e se o número MAX-TRIES é atingido, é retornada uma falha.

Como explicado por Kautz, Selman e Mcallester (2004), o procedimento consiste em:

1. Gerar uma atribuição aleatória para as variáveis;
2. verificar se a atribuição atual satisfaz a função, se sim, retornar a solução;
3. escolher uma cláusula não satisfeita C aleatoriamente;
4. se existe uma variável em C com $BreakCount = 0$ é feito um *flip* nessa variável e depois voltar ao passo 2;
5. - probabilidade p de fazer um *flip* em uma variável em C escolhida aleatoriamente;
   - probabilidade $1-p$ de fazer um *flip* na variável em C com o menor *breakCount*;
6. se tiver sido atingido o número máximo de iterações, voltar para o passo 2;
7. se tiver sido atingido o número máximo de tentativas, retornar falha.

A grande vantagem do Walksat é a utilização de ruído, já que dá ao algoritmo capacidade de realizar muitas operações com menos tempo e, mesmo quando realizados os movimentos aleatórios (*random walk moves*), é garantido que pelo menos uma cláusula se torne verdadeira em cada iteração do procedimento.

4 Implementação do Walksat em GPU

A construção de Resolvedores SAT que implementam paralelização tem grande potencial de resultar em desempenho substancialmente melhor do que modelos puramente seriais. Por esse motivo, Resolvedores SAT paralelos são importante tópico de discussão na comunidade de Satisfatibilidade no momento, como por exemplo em nos estudos de Osama, Wijs e Biere, (2021) e Deleau, Jaillet e Krajecki (2008).

Devido à grande quantidade de áreas em que pode se aplicar SAT, melhoras no desempenho de resolvedores são muito importantes. Solvers com melhor desempenho podem ser capazes de resolver fórmulas muito mais complexas do que é possível hoje em tempo aceitável.

Tendo em vista o menor desempenho de GPUs em relação a CPUs na realização de processamento puramente serial e nas transferências de dados necessárias para o uso de aceleração de hardware, algoritmos que usam aceleração de GPU podem ter desempenho igual ou pior do que se mantivessem seu processamento inteiramente em CPU, quando não são capazes de utilizar apropriadamente a larga capacidade de paralelização das GPUs.

Considerando o principal objetivo deste trabalho, que é investigar o potencial de melhora de um Resolvedor SAT baseado no algoritmo Walksat ao se aplicar processamento paralelo, apresenta-se, neste capítulo, como resultado deste trabalho, uma análise do desempenho paralelo do Walksat, como ele se relaciona com seu desempenho serial, os fatores que afetam esse desempenho, bem como dois Resolvedores SAT, baseados no Walksat, um serial e um paralelo em GPU, que podem ser usados para estudos futuros.

4.1 Metodologia

Para que seja possível fazer uma avaliação justa da diferença de desempenho entre uma implementação serial e uma paralela do Walksat, é necessário que a própria paralelização seja a única variável que mude entre as duas implementações. Para isso, antes de ser desenvolvida a aplicação paralela, foi feita uma implementação serial do Walksat, chamada de SerialWalksat, para servir de controle.

Esta implementação foi feita a partir da implementação mais recente do Walksat (Walksat_v56), disponibilizada pelo seu criador, Henry Kautz (Kautz, Selman e Mcallester, 2004), aqui chamada de OriginalWalksat. É importante notar que esta implementação mais recente contém uma série de otimizações e técnicas não descritas em 2 por Kautz, Sabharwal e Selman (2008). Algumas dessas técnicas foram incorporadas no SerialWalksat e a diferença de desempenho que poderá ser observada mais adiante, entre o

OriginalWalksat e o SerialWalksat, deve-se àquelas que não foram incorporadas.

O algoritmo paralelo, chamado aqui de CudaWalksat, foi criado a partir do SerialWalksat. Isso significa que todas as operações (*flips*, cálculo de *BreakCounts* etc) realizadas no conjunto de dados são as mesmas quando a mesma semente é fornecida à função `srand()`[1].

4.2 Metodologia de paralelização do Walksat

No Algoritmo 2, para que seja realizada a operação da linha 6, é necessário que seja calculado quais cláusulas não estão satisfeitas. Da mesma forma, a operação da linha 7 requer que sejam calculados os *BreakCounts* de todas as variáveis contidas na cláusula escolhida. Esses dois cálculos são as operações que mais requerem processamento no Walksat.

As funções `updateTrueInClausesAfterFlip` e `calculateBreakCountAfterFlip`, que realizam os cálculos descritos acima, foram transformadas em funções paralelas que se executam em GPU e cujo tempo de processamento está representado na Figura 4 nas cores roxo e verde respectivamente. A versão paralela destas duas funções está nos Códigos 3 e 4

Nas linhas 7 e 13 do Algoritmo 2, também é necessário processamento para encontrar a variável com o menor *BreakCount* em C. Essa operação se resume em encontrar o menor valor em um array unidimensional, o que é feito serialmente.

O processamento da linha 13 poderia ser feito serialmente em GPU ou em CPU. A segunda opção implicaria que o vetor que armazena os *BreakCounts* precisaria ser copiado para a memória principal.

Foi escolhida a realização dessa operação em CPU, porque o processamento somado da cópia de memória e do processamento em CPU foi menor do que o do processamento em GPU dessa operação na máquina descrita em 4.4.

4.3 Implementação paralela

Depois de desenvolvido o SerialWalksat, seu código foi usado como base para a implementação do CudaWalksat, de maneira que quando usada uma mesma semente na função `srand()`, as operações realizadas no conjunto de dados pelos dois resolvedores sejam as mesmas.

---

[1] Função que define quais números aleatórios serão gerados pela função `rand()` dependendo da semente recebida.

O uso do NVVP (2.4.2.4.2) foi crucial para a otimização da implementação CudaWalksat pois forneceu importantes informações, as quais permitiram o melhoramento do desempenho do resolvedor.

4.3.1 Versão Inicial

O primeiro solver baseado no SerialWalksat foi implementado e se mostrou funcional. Entretanto, estava claro que não tinha sido atingida a melhor implementação possível e outras escolhas levariam a um resultado melhor.

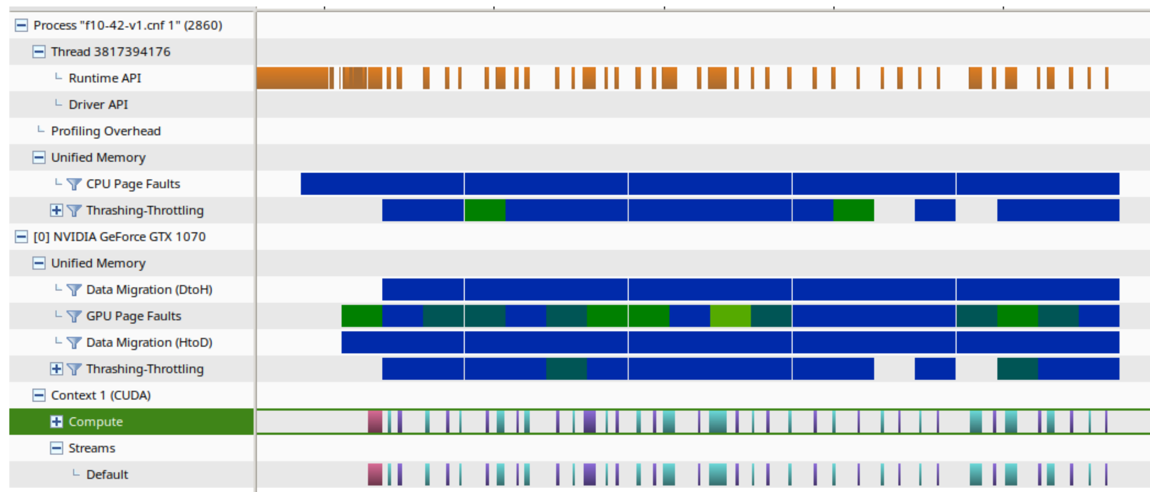

Figura 3 – NVVP implementação otimizada em CUDA

A Figura 3 mostra o resultado apresentado pelo NVVP para uma execução dessa versão inicial do algoritmo paralelo. Está demonstrado por meio de barras, que representam tempo, quanto processamento foi gasto em qual atividade. Observando que as barras da linha *Unified Memory* se extendem a quase toda a execução do programa e que as barras da linha *Compute* ocupam bem menos espaço no gráfico, fica claro que a maior parte do tempo de processamento é utilizado somente em transferências de dados entre a memória principal e a da GPU.

O mal desempenho da aplicação e a quantidade de tempo gasto em transferências de dados foi resultado, principalmente, do mal uso das ferramentas de memória fornecidas pelo CUDA. Nesta versão, estava sendo usada memória unificada sem que houvesse a necessidade, porque não houve reflexão sobre quando estariam sendo realizadas as transferências de dados. Teorizou-se que melhor desempenho poderia ser alcançado seguindo as instruções de Harris (2017).

4.3.2 Otimizações

A paralelização do algoritmo foi refeita, desde o começo com a inteção de minimizar as transferências de dados entre memória principal e GPU. Isto foi atingido por meio da

minimização do processamento feito em CPU, para que seja mínima a necessidade de dados atualizados estarem presentes na memória principal e o deslocamento de atividades que seriam realizadas no CPU para a GPU.

Um importante fator levado em conta, durante essa implementação, foi a comparação entre o tempo necessário para fazer atividades necessariamente seriais na GPU e o tempo somado entre copiar os dados necessários para a memória principal e a realização dessas atividades em CPU.

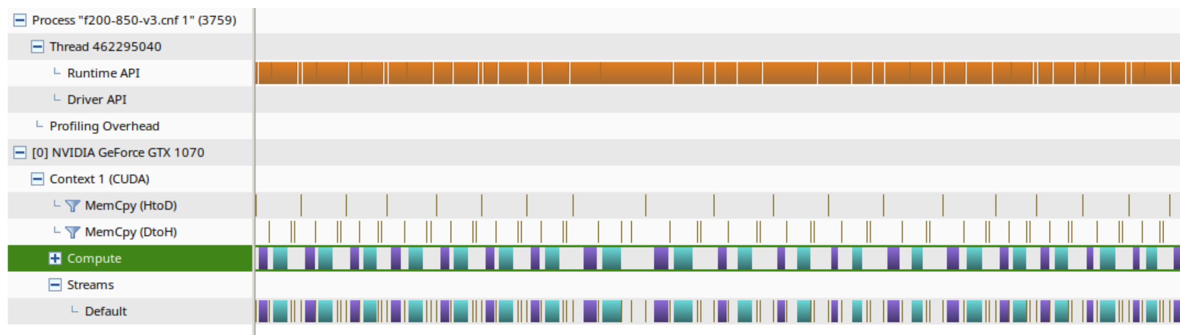

Figura 4 – NVVP primeira implementação paralela em CUDA

O *Nvidia Visual Profiler* foi usado para avaliar o desempenho dessa implementação também. Na Figura 4 é possível notar que a cópia de dados, representadas pelas linhas *MemCpy(HtoD)* e *MemCpy(DtoH)*, não é mais o principal gargalo de processamento e que o tempo gasto pelo programa é majoritariamente usado no processamento pela GPU, representado pelas barras coloridas da linha *Compute* e no pouco processamento que ocorre em CPU que, somado com as chamadas à api CUDA, aparece como espaços em branco na linha *Default*.

Como no CudaWalksat só são feitas transferências de dados explicitamente e todas as que são realizadas são necessárias para o funcionamento do algoritmo, demonstrando-se que tempo de processamento é desperdiçado, por não ser possível usar uma porção maior do tempo de processamento para realizar computação na GPU. Além disso, um algoritmo diferente do Walksat, com operações mais paralelizáveis, ou uma diferente implementação paralela do Walksat, poderiam melhor utilizar esse tempo.

4.3.3 Algoritmo Paralelo

O CudaWalksat é apresentado nesta seção por meio de códigos simplificados da implementação prática do Algoritmo 2.

O Código 1 é a função `main` do CudaWalksat e representa a lógica do Algoritmo 2. Nela é chamada a função `flipProcedure`, listada no Código 2, que realiza um *flip*, assim como os cálculos que precisam ser feitos para que o próximo *loop* da função *main* possa ocorrer.

```
THREADS_PER_BLOCK = stoi(argv[2]);

int seed = ((tv.tv_sec & 0177) * 1000000) + tv.tv_usec;
srand(seed);

getCNFFromStdin(argv[1]);
fillOccurenceList();

for (triesCounter in MAX_TRIES) {
  generateRandomAssignment();
  countTrueInClauses();
  calculateAllBreakCounts();

  for (flipsCounter in MAX_FLIPS) {

    if (numberOfUnsatClauses == 0) {
      printSucessfulResult();
      return 0;
    }
    randomClause = chooseRandomUnsatisfiedClause();

    variableWithSmallestBreak = getSmallestBreakcountInClause(randomClause);

    if (breakCount[variableWithSmallestBreak] == 0 &&
        makeCount[variableWithSmallestBreak] > 0) {
      flipProcedure(variableWithSmallestBreak);
    } else {
      probability = rand() % 100 + 1;

      if (probability >= NOISE) {
        flipProcedure(ABS(
            clauses[randomClause][rand() % (*clauseSizes)[randomClause]]));
      } else {
        flipProcedure(variableWithSmallestBreak);
      }
    }
  }
}
cout << "Failed" << endl;
return 0;
```

Código 1: CudaWaksat Main

No Código 2 é chamada a função `flip`, que inverte o valor de uma variável na memória, e então são feitas as chamadas os `kernels`, usando a variável `THREADS_PER_BLOCK`, definida em tempo de execução. Além disso são requisitadas as transferências de dados entre GPU e memória principal.

Os *kernels* `updateTrueInClausesAfterFlip` e `calculateBreakCountAfterFlip` são chamados na função `flipProcedure`, e estão listados nos Códigos 3 e 4. Estes *kernels* compõem a maior parte da computação feita pelo CudaWalksat por tempo e são as partes cujo processamento é feito em GPU.

Nos *kernels*, para modificar dados na memória da GPU, é usada a função `atomicAdd`, que abstrai ao programador o uso de semáforos.

O *kernel* calculateBreakCountAfterFlip faz uso das funções listadas no Código 5, sendo que cada uma possui um *loop* `for` que deve ser executado em cada instância do *kernel*.

```
void flipProcedure(int variable) {
  int N = numberOfOccurences[variable];

  flip(variable);

  cudaMemcpy(variables, cudaMemcpyHostToDevice);
  int blocks = ((*numberOfOccurences)[variable] + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

  updateTrueInClausesAfterFlip<<<blocks, THREADS_PER_BLOCK>>>
                                 (variable, numberOfOccurences[variable]);

  cudaMemcpy(trueInClause, cudaMemcpyDeviceToHost);

  calculateBreakCountAfterFlip<<<blocks, THREADS_PER_BLOCK>>>
                                 (variable, numberOfOccurences[variable]);

  cudaMemcpy(breakCount, cudaMemcpyDeviceToHost);
  cudaMemcpy(makeCount, cudaMemcpyDeviceToHost);
  cudaMemcpy(numberOfUnsatClauses, cudaMemcpyDeviceToHost);

}
```

Código 2: CudaWaksat flipProcedure

```
__global__ void updateTrueInClausesAfterFlip_CUDA(int variable, int n) {
  int gid = threadIdx.x + blockIdx.x * blockDim.x;
  if (gid < n) {

    int trueInClauseBeforeFlip = trueInClause[occurences[variable][gid]];
    int trueInClauseAfterFlip = countTrueInClause(occurences[variable][gid]));

    if (trueInClauseBeforeFlip != trueInClauseAfterFlip) {
      if (trueInClauseBeforeFlip == 0) {
        atomicAdd(numberOfUnsatClauses, -1);
      } else if (trueInClauseAfterFlip == 0) {
        atomicAdd(numberOfUnsatClauses, 1);
      }
    }
  }
}
```

Código 3: CudaWaksat updateTrueInClausesAfterFlip_kernel

4.4 Resultados

Para avaliar o desempenho de CudaWalksat em comparação com SerialWalksat e OriginalWalksat, cada algoritmo foi executado para o mesmo conjunto de fórmulas Booleanas utilizando a mesma semente `S = 89197532` para consistência de resultados entre testes.

Nesta seção serão apresentados os resultados dos testes, sobre os quais, são ressaltadas as seguintes informações relativas aos procedimentos utilizados:

- Todos os testes foram realizados em um computador com um CPU Intel i5 6600k a 3.5Ghz, GPU Nvidia GTX 1070 e memória RAM configurada a 2100 MT/s usando sistema operacional Arch Linux com kernel versão 5.19.3-arch1-1;

```
__global__ void calculateBreakCountAfterFlip_CUDA(int variableNumber, int n) {
  int gid = threadIdx.x + blockIdx.x * blockDim.x;
  if (gid < n) {

    int thisClause = occurences_gpu[variableNumber][gid];

    bool previousValueInClause = (!getValue(variableNumber)) == (thisClause > 0);
    thisClause = ABS(thisClause);
    if (!previousValueInClause) {
      switch (trueInClause_gpu[thisClause]) {
      case 1:
        atomicAdd(breakCount_gpu[variableNumber], 1);
        decreaseMakeCountInClause(thisClause);
        break;
      case 2:
        atomicAdd(breakCount_gpu[variableNumber], 1);
        decreaseBreakCountInClause(thisClause);
        break;
      }
    } else {
      switch (trueInClause_gpu[thisClause]) {
      case 0:
        atomicAdd(breakCount_gpu[variableNumber], -1);
        increaseMakeCountInClause(thisClause);
        break;
      case 1:
        increaseBreakCountInClause(thisClause);
        break;
      }
    }
  }
}
```

Código 4: CudaWaksat calculateBreakCountAfterFlip

- foi usado como medida de desempenho, o tempo gasto para encontrar uma solução para cada fórmula;
- o valor em segundos considerado para cada algoritmo em cada fórmula foi a média aritmética dos tempos de 10 execuções para cada fórmula;
- o tempo de cada execução foi medido com a ferramenta `time` incluída com GNU/-Linux;
- no programa OriginalWalksat foram usados os seguintes argumentos: `-numsol 1 -seed 89197532`. Assim, sua execução para quando for encontrada uma solução para a fórmula e é definida a semente usada;
- o uso da mesma semente para CudaWalksat e para SerialWalksat significa que os dois vão realizar as mesmas operações sobre o conjunto de dados. Não necessariamente o OriginalWalksat vá realizar as mesmas operações que os outros dois;
- considerando as variações entre os algoritmos SerialWalksat e OriginalWalksat, os efeitos da paralelização em si podem ser observados pela diferença de desempenho entre SerialWalksat e CudaWalksat;

```
__device__ void increaseMakeCountInClause(int clause) {
  for (i in clauseSizes[clause]) {
    if (!clauses[clause][i]) {
      atomicAdd(makeCount[clauses)[clause][i])], 1);
    }
  }
}

__device__ void decreaseMakeCountInClause(int clause) {
  for (i in clauseSizes[clause]) {
    if (!clauses[clause][i]) {
      atomicAdd(makeCount[clauses[clause][i])], -1);
    }
  }
}

__device__ void increaseBreakCountInClause(int clause) {
  for (i in clauseSizes[clause]) {
    if (clauses[clause][i]) {
      atomicAdd(breakCount[clauses[clause][i])], 1);
    }
  }
}

__device__ void decreaseBreakCountInClause(int clause) {
  for (i in clauseSizes[clause]) {
    if (clauses[clause][i]) {
      atomicAdd(breakCount[clauses[clause][i])], -1);
    }
  }
}
```

Código 5: CudaWaksat BreakCount Functions

- embora não haja limitação de tamanho de cláusulas no uso dos algoritmos, todas as fórmulas tinham 3 variáveis em todas as cláusulas.

4.4.1 Desempenho em relação ao número de cláusulas e variáveis

Para avaliação do desempenho dos programas em relação ao número de cláusulas e variáveis, foi usado o seguinte conjunto de fórmulas:

- 1 fórmula com 3 cláusulas e 4 variáveis;
- 1 fórmula com 3 cláusulas e 2 variáveis;
- 1 fórmula com 10 cláusulas e 42 variáveis;
- 4 fórmulas com 200 cláusulas e 850 variáveis;
- 1 fórmula com 300 cláusulas e 1275 variáveis;
- 2 fórmulas com 400 cláusulas e 1700 variáveis;
- 5 fórmulas com 800 cláusulas e 3400 variáveis;

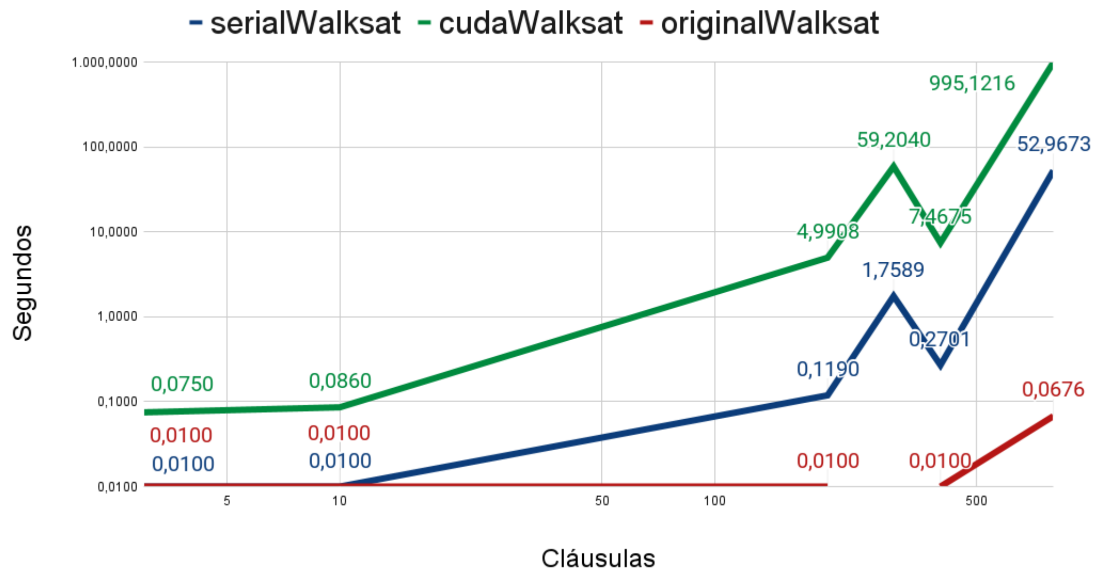

Figura 5 – Desempenho dos algoritmos por cláusulas (escala logarítmica)

Número de Cláusulas	SerialWalksat	CudaWalksat	OriginalWalksat
3	0,0100	0,0750	0,0100
10	0,0100	0,0860	0,0100
200	0,1190	4,9908	0,0100
300	1,7589	59,2040	-
400	0,2701	7,4675	0,0100
800	52,9673	995,1216	0,0676

Tabela 1 – Desempenho dos algoritmos por cláusulas

Nos testes cujos resultados estão nas tabelas 1 e 2 e nas Figuras 5 e 6, consistentemente, CudaWalksat obteve um desempenho pior do que SerialWalksat independentemente do número de cláusulas ou variáveis. No teste com 300 cláusulas não existe dados para o OriginalWalksat pois não obteve solução. Por isso nas Tabelas 1 e 2 e nas Figuras 5 e 6 existe um teste com falta de dados para o OriginalWalksat.

Como demonstrado pelos dados, o tamanho das fórmulas (número de variáveis e de cláusulas) afeta o desempenho de todos os três algoritmos mas o tempo de execução de CudaWalksat cresce especialmente rapidamente.

4.4.2 Desempenho em relação à razão de variáveis por cláusulas

Para avaliar o desempenho dos algoritmos quando é variada a proporção entre variáveis e cláusulas em cada fórmula, foi usado o seguinte conjunto de fórmulas, no qual

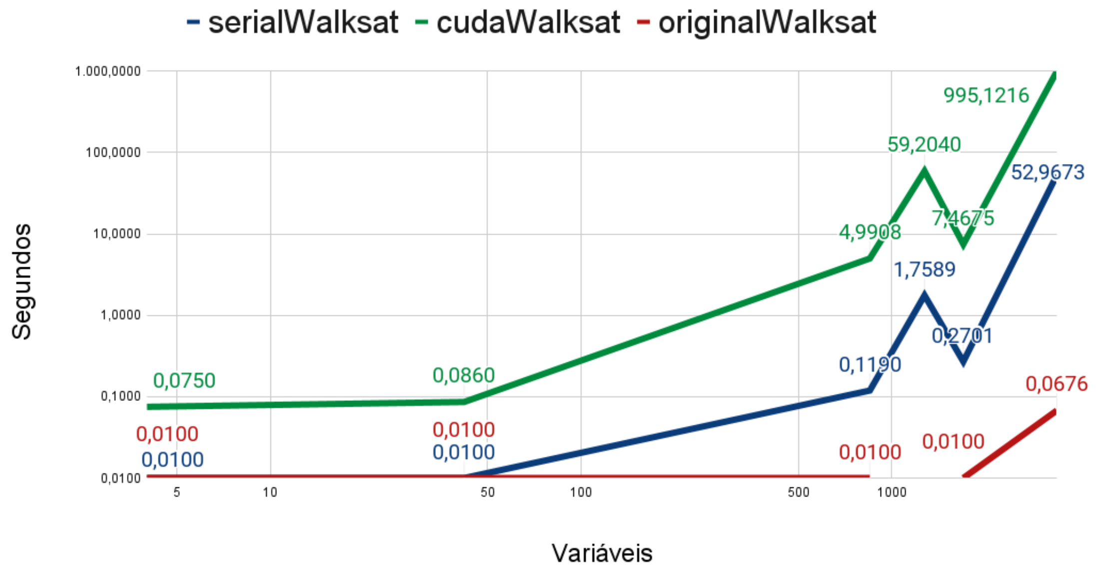

Figura 6 – Desempenho dos algoritmos por variáveis

Número de Variáveis	SerialWalksat	CudaWalksat	OriginalWalksat
4	0,0100	0,0750	0,0100
42	0,0100	0,0850	0,0100
850	0,0905	3,5875	0,0100
1275	1,7585	58,2750	-
1700	0,2685	7,3375	0,0100
3400	12,9000	231,7600	0,0300

Tabela 2 – Desempenho dos algoritmos por variáveis

a razão de variáveis por cláusulas varia de 3,75 a 4,25, intervalo que foi escolhido porque todos os algoritmos falharam em conseguir uma solução quando a proporção é maior que 4,25:

- 2 fórmulas com 800 cláusulas e 3000 variáveis;
- 2 fórmulas com 800 cláusulas e 3200 variáveis;
- 2 fórmulas com 800 cláusulas e 3400 variáveis;

Pela Figura 7, percebe-se que em nenhuma proporção de variáveis por cláusulas CudaWalksat se torna mais performático do que os outros dois algoritmos.

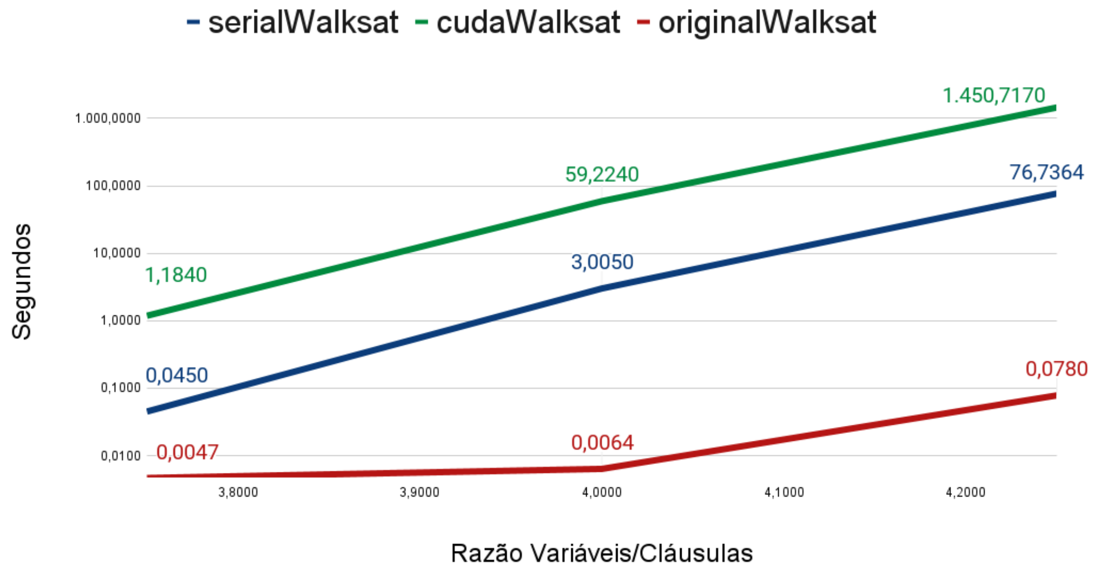

Figura 7 – Desempenho dos algoritmos por razão (escala logarítmica)

Razão Variáveis/Cláusulas	SerialWalksat	CudaWalksat	OriginalWalksat
3,75	0,0450	1,1840	0,0047
4,00	3,0050	59,2240	0,0064
4,25	76,7364	1.450,7170	0,0780

Tabela 3 – Desempenho dos algoritmos por razão

4.4.3 Análise

É demonstrado pelos dados que CudaWalksat tem desempenho pior do que SerialWalksat.

A análise do CudaWalksat feita pelo NVVP na Figura 3 demonstra que, uma porção muito pequena do tempo de processamento é gasta com transferências de memória e que a maior parte do tempo de processamento gasto está sendo feito em GPU. Portanto, estes não são os principais gargalos de processamento.

Também fica claro, pela Figura 3 e pela própria arquitetura do algoritmo paralelo, que pode ser vista no Código 2, que das cópias de memória e a execução dos *kernels*, somente uma é realizada de cada vez, sendo que a tecnologia permite que mais de um *kernel* seja executado de cada vez e que cópias de dados sejam feitas em paralelo à execução de kernels. Teoriza-se que, com a otimização desses pontos, o desempenho do CudaWalksat poderia ser melhor.

Como explicado na Seção 4.2, assim como pode ser observado no Código 2, cada

função paralela tem como número máximo de instâncias o número de ocorrências da variável em que foi feito o último *flip*, número que é obrigatoriamente menor ou igual ao número de cláusulas da função. Esta limitação, agregada ao fato que somente um *kernel* roda de cada vez, significa que o número de instâncias sendo processadas não pode ser a soma dos dois *kernels* e que são feitos *loops* dentro dos *kernels*, demonstrados nos Códigos 4 e 3, revela que o CudaWalksat não é capaz de utilizar todo o potencial de processamento paralelo da GPU.

5 Conclusão

Para o alcance dos objetivos deste trabalho fora realizados testes comparativos entre a versão original do Walksat com processamento serial e duas implementações do Walksat desenvolvidas como parte deste trabalho, uma serial e outra paralela em GPU.

Os resultados obtidos a partir do teste e comparação dessas três implementações demonstram que a versão em GPU tem desempenho pior que suas alternativas seriais em todos os cenários, mesmo que seja variado o tamanho da fórmula testada ou a proporção de variáveis para cláusulas.

A análise feita na Subseção 4.4.3 demonstrou que o CudaWalksat não foi capaz de utilizar toda a capacidade de paralelização da GPU. Portanto teoriza-se que com melhor otimização do programa paralelo, melhor desempenho seria alcançado.

Devido ao número relativamente baixo de instâncias paralelas que podem ser criadas pelo CudaWalksat, explicadas na Seção 4.2, devido aos fatores colocados na Seção 1.4, teoriza-se que uma implementação do CudaWalksat feita para usar processamento paralelo em CPU poderia ter desempenho melhor do que o CudaWalksat e o SerialWalksat.

Como trabalho futuro, vislumbra-se o estudo mais aprofundado sobre maneiras de utilizar as capacidades paralelas das GPUs no Walksat, uma análise mais profunda dos dados das ferramentas de *profiling* da Nvidia para que seja possível identificar os pontos críticos de melhora para a implementação paralela e a realização de uma implementação do CudaWalksat utilizando processamento paralelo em CPU.

Referências

BRITANNICA. Np-complete problem. 325 North LaSalle Street Suite 200 Chicago, IL 60654 USA, 2022. Citado na página 14.

DELEAU, H.; JAILLET, C.; KRAJECKI, M. Gpu4sat: solving the sat problem on gpu. Université de Reims Champagne-Ardenne, 2008. Citado 2 vezes nas páginas 11 e 24.

ESCOFFIER, B.; PASCHOS, V. The np-completeness column. HAL open science, 2006. Citado na página 14.

FRANCO, J.; MARTIN, J. *A History of Satisfiability*. [S.l.]: IOS Press, 2009. <http://gauss.ececs.uc.edu/SAT/articles/FAIA185-0003.pdf>. Citado 2 vezes nas páginas 9 e 13.

HARRIS, M. Using shared memory in cuda c/c++. 1087 Budapes Luther utca 4. 5. em. 36., 2013. Citado na página 21.

HARRIS, M. Unified memory for cuda beginners. 1087 Budapes Luther utca 4. 5. em. 36., 2017. Citado na página 26.

INTEL. Opencl™ 2.0 shared virtual memory overview. 2200 Mission College Blvd, Santa Clara, CA 95054, United States, 2014. Citado na página 21.

KAUTZ, H.; SABHARWAL, A.; SELMAN, B. *Incomplete Algorithms*. [S.l.]: IOS Press, 2008. <https://www.cs.cornell.edu/~sabhar/chapters/IncompleteAlg-SAT-Handbook-prelim.pdf>. Citado 7 vezes nas páginas 9, 10, 14, 15, 16, 22 e 24.

KAUTZ, H.; SELMAN, B.; MCALLESTER, D. A. Walksat in the 2004 sat competition. University of Rochester, 2004. Citado 3 vezes nas páginas 22, 23 e 24.

KLEMENT, K. C. Propositional logic. Estados Unidos da América, 2022. Citado na página 13.

LARROSA, J.; LYNCE, I.; MARQUES-SILVA, J. Satisfiability: Algorithms, applications and extensions. Universitat Polit´ecnica de Catalunya and Technical University of Lisbon and 3University College Dublin, 2010. Citado na página 13.

LEE; ROYCHOWDHURY; SESHIA. *Fundamental Algorithms for System Modeling, Analysis, and Optimization*. UC Berkeley: [s.n.], 2011. <https://ptolemy.berkeley.edu/projects/embedded/eecsx44/fall2011/lectures/SATSolving.pdf>. Citado na página 9.

LIU, J.; HU, F. Q.; LI, X. Performance comparison on parallel cpu and gpu algorithms for unified gas-kinetic scheme. School of Energy and Power Engineering, Beihang University, Beijing and Department of Mathematics and Statistics, Old Dominion University, Norfolk, 2018. Citado na página 11.

MARQUES-SILVA, J. Practical applications of boolean satisfiability. IEEE, 2008. Citado 2 vezes nas páginas 9 e 14.

MATTSON, T.; SANDERS, B.; MASSINGILL, B. *Berna Massingill*. E301 CSE Building, P.O. Box 116120, Gainesville, FL 32611: University of Florida, 2005. <https://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ParallelHardware.htm>. Citado na página 16.

MCDONALD, A. Parallel walksat with clause learning. Carnegie-Mellon University, 2009. Citado na página 10.

MEMETI, S. et al. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In: *ARMS-CC@PODC*. [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 18 e 19.

NVIDIA. *CUDA C++ Programming Guide*: 4. hardware implementation. 2788 San Tomas Expy, Santa Clara, CA 95051, United States, 2022. Citado 3 vezes nas páginas 17, 19 e 21.

NVIDIA. *HPC SDK Documentation*. 2788 San Tomas Expy, Santa Clara, CA 95051, United States, 2022. Citado na página 18.

NVIDIA. *NVIDIA Visual Profiler*. 2788 San Tomas Expy, Santa Clara, CA 95051, United States, 2022. Citado na página 20.

OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Programming Interface*. [S.l.], 2018. Citado na página 18.

OSAMA, M.; WIJS, A.; BIERE, A. Sat solving with gpu accelerated inprocessing. Eindhoven University of Technolog and Johannes Kepler University, 2021. Citado 2 vezes nas páginas 11 e 24.

RASTERGRID. Simd in the gpu world. 1087 Budapes Luther utca 4. 5. em. 36., 2022. Citado na página 17.

SINZ, C. *Practical Applications of SAT*. Johannes Kepler University Linz: [s.n.], 2005. <https://www.carstensinz.de/talks/RISC-2005.pdf>. Citado na página 9.