PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

João Paulo Pereira Novais

Uma Abordagem de Arquitetura Paralela para Suporte na Extração de Conceitos Formais

Belo Horizonte

2020

João Paulo Pereira Novais

Uma Abordagem de Arquitetura Paralela para Suporte na Extração de Conceitos Formais

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. Henrique Cota de Freitas

Coorientador: Prof. Dr. Mark Alan Junho Song

Belo Horizonte

2020

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Novais, João Paulo Pereira
N935a Uma abordagem de arquitetura paralela para suporte na extração de conceitos formais / João Paulo Pereira Novais. Belo Horizonte, 2020.
71 f. : il.

Orientador: Henrique Cota de Freitas
Coorientador: Mark Alan Junho Song

Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática

1. Computação de alto desempenho. 2. Conceitos - Análise. 3. Mineração de dados (Computação). 4. Algoritmos computacionais. 5. Arquitetura de computador. 6. Processamento paralelo (Computadores). I. Freitas, Henrique Cota de. II. Song, Mark Alan Junho. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.

CDU: 681.3-11

Ficha catalográfica elaborada por Fabiana Marques de Souza e Silva - CRB 6/2086

João Paulo Pereira Novais

Uma Abordagem de Arquitetura Paralela para Suporte na Extração de Conceitos Formais

Dissertação apresentada ao Programa de Pós-graduação em Informática como requisito parcial para qualificação ao grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais.

Prof. Dr. Henrique Cota de Freitas - PUC Minas
(Orientador)

Prof. Dr. Mark Alan Junho Song - PUC Minas
(Coorientador)

Prof. Dr. Luis Enrique Zárate - PUC Minas
(Banca Examinadora)

Prof. Dr. Mario Antonio Ribeiro Dantas - UFJF
(Banca Examinadora)

Belo Horizonte, 2 de junho de 2020.

A Deus, meus pais, família, amigos e à Tiza

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por todas as bençãos concedidas, sejam elas estar com pessoas que amo, crescer com sabedoria, ter capacidade de superar minhas dificuldades, e conquistar minhas metas. Aos meus heróis, Margarida e Miguel, que também chamo de pais, agradeço por tudo... desde me criarem e me transformarem em tudo que sou hoje, por me darem amor, carinho, estarem sempre ao meu lado por quaisquer motivos que eu preciso, ou até mesmo quando não preciso. Ao meu grande amigo Lucas Andrade (Nerd), agradeço por ter sido presente nessa jornada acadêmica cheia de percalços, mas também recheada de descontrações e perseverança. Ao meu irmão de consideração, Lucas Paulinelli, obrigado pelos momentos de descontração e por me apoiar sempre que preciso. Aos meus familiares por todo o apoio. E finalmente, a Tiza, minha companheira, obrigado por me amar, estar sempre ao meu lado quando preciso, me levantar e me apoiar nos momentos de desanimo, me ouvir e acreditar nos meus sonhos. Amo todos vocês!

Agradecimento especial ao Prof. Dr. Henrique Cota de Freitas, meu orientador, que me instruiu impecavelmente na minha jornada acadêmica, desde a graduação até o mestrado. Agradeço por todas as orientações, instruções, conversas e compreensões. Sou muito grato por todos os ensinamentos que contribuíram para minha formação. Meu sentimento é de sorte por ter sido seu orientando. Deixo também os agradecimento ao meu coorientador Prof. Dr. Mark Alan Junho Song e a todos os professores da graduação e mestrado que compartilharam seus conhecimentos ajudando a alcançar minhas metas e sonhos.

Aos amigos e colegas do grupo CArT (Computer Architecture and Parallel Processing Team), agradeço pelos momentos de discussões e trocas de experiência. Em especial, agradeço ao Matheus, que tanto me ajudou nesta caminhada sempre disposto em ajudar em quaisquer problemas.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), agradeço pela bolsa e confiança concedida, além da colaboração para o desenvolvimento acadêmico do país. Agradecimentos também ao CNPq, FAPEMIG, PUC Minas e Intel Hardware Accelerator Research Program (HARP) pelo suporte no desenvolvimento do trabalho.

Por fim, agradeço aos meus amigos que fizeram parte dessa jornada que se finda, e ainda farão parte das outras que estão por vir.

"O impossível existe até que alguém duvide dele e prove o contrário."

Albert Einstein

RESUMO

A crescente complexidade de problemas reais direcionou a atenção de muitas pesquisas que lidam com grandes volumes de dados. Como exemplo, a Análise Formal de Conceitos que, por ser considerada uma teoria importante para formalizar a representação do conhecimento, tem sido cada vez mais explorada. Essa teoria utiliza estrutura conceitual para organizar hierarquicamente conceitos de um contexto formal que consiste em objetos, atributos e suas incidências. No entanto, contextos formais com alta dimensionalidade demandam um excessivo recurso computacional. Diversos algoritmos foram propostos para extrair conceitos formais com complexidades exponenciais no pior caso, o que os torna impraticáveis em situações de alta dimensionalidade. Esse problema motivou vários estudos que buscam alternativas para lidar com esse tipo de cenário. Nesta dissertação, arquiteturas paralelas (compostas por CPU, FPGA e GPU) são propostas e avaliadas para extração de conceito formal, utilizando um algoritmo Força Bruta desenvolvido em OpenCL. A proposta CPU + GPU apresenta um desempenho e escalabilidade mais alto ao processar contextos de alta dimensionalidade com um grande número de objetos e atributos, mesmo em comparação com outros algoritmos. Os resultados apresentam um desempenho até 18x maior que um algoritmo mais inteligente denominado Data-Peeler. Além disso, possui maior eficiência energética, atingindo pelo menos 1,79x mais operações por consumo de energia em comparação a outros algoritmos explorados neste trabalho.

Palavras-chave: Algoritmo Força Bruta, OpenCL, Arquiteturas Heterogêneas, Análise Formal de Conceitos

ABSTRACT

The growing complexity of real problems has drawn the attention of many researches dealing with large volumes of data. For instance, Formal Concept Analysis is considered an important theory to formalize knowledge representation, and it has been increasingly explored. This theory uses a conceptual structure to hierarchically organize concepts from a formal context that consists of objects, attributes and their incidences. However, formal contexts with high dimensionality demand an excessive computational resource. Several algorithms have been proposed to extract formal concepts with exponential complexities in the worst case, which makes them impractical in situations of high dimensionality. This problem has motivated several studies looking for alternatives to deal with this type of scenario. In this dissertation, parallel architectures (composed of CPU, FPGA and GPU) are proposed and evaluated for the extraction of a formal concept, using an OpenCL-based Brute Force algorithm. The CPU + GPU proposal presents a higher performance and scalability when processing high dimensional contexts with a large number of objects and attributes, even in comparison with other algorithms. The results show a performance up to 18x higher than a more intelligent algorithm called Data-Peeler. In addition, it has greater energy efficiency, reaching at least 1.79x more operations per energy consumption compared to other algorithms explored in this work.

Keywords: Brute Force Algorithm, OpenCL, Heterogeneous Architectures, Formal Concept Analysis

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BC *Bloco de Conceito*

BDD *Binary Documents Diagrams*

BDD *Binary-Decision Diagrams*

CbO *Close-by-One*

CPU *Central Processing Unit*

FCA *Formal Concept Analysis*

FCbO *Fast Close-by-One*

FLC *Fuzzy logic controller*

FPGA *Field Programmable Gate Array*

GPUs *Graphics Processing Units*

ICG *Incidence Combination Generator*

MIMD *Multiple instructions, multiple data*

MISD *Multiple instructions, single data*

MOPS *Milhões de operações por segundo*

RAM *Random Access Memory*

ROM *Read Only Memory*

SCGaz *Synthetic Context Generator*

SIMD *Single instruction, multiple data*

SISD *Single instruction, single data*

TFCA *Triadic Formal Concept Analysis*

UART *Universal Asynchronous Receiver-Transmitter*

USB *Universal Serial Bus*

VHDL *VHSIC Hardware Description Language*

SUMÁRIO

1 INTRODUÇÃO

Várias pesquisas atuais têm como objetivo encontrar padrões a partir de conjuntos de dados em áreas como marketing, medicina e redes sociais. Para realizar essas pesquisas, algumas abordagens são utilizadas: aprendizado de máquina, inteligência artificial, mineração de dados e análise formal de conceitos (GANTER; RUDOLPH; STUMME, 2019; JALALI et al., 2017). Essas abordagens têm se tornado cada vez mais relevantes para a comunidade científica para identificar padrões ou extrair conhecimento de grandes conjuntos de dados.

Dentre as diversas áreas para extração de conhecimento, a análise formal de conceitos busca identificar o significado dos comportamentos através de uma tabela de dados, transformando-as em estruturas algébricas. Essas tabelas são representadas por uma relação entre os objetos e atributos, que resultam nas estruturas algébricas, podendo ser utilizadas para visualização e interpretação dos dados.

Ciente da relevância de identificar comportamentos através de um conjunto de dados, a área de FCA (do inglês *Formal Concept Analysis*) tem sido amplamente estudada para identificar padrões em conjuntos binários (MISSAOUI; KUZNETSOV; OBIEDKOV, 2017). FCA é um método para representar dados de maneira formal, podendo ser aplicado a muitas áreas para extrair padrões de dados, por exemplo, em redes sociais (HAO et al., 2016). Devido à importância da descoberta de conhecimento, algoritmos como *In-Close* (ANDREWS; ORPHANIDES, 2010) e *Data-Peeler* (CERF et al., 2008), visam processar e extrair conhecimento dos conjuntos de dados. Além disso, esses algoritmos têm a capacidade de reestruturar conjuntos de dados para minimizar comparações desnecessárias em busca de mais desempenho. No entanto, lidar com grandes quantidades de dados em tempo reduzido é um grande desafio.

Nesse cenário, é necessário melhorar o desempenho do processamento e reduzir o consumo de energia dos processadores. Para isso, são necessários sistemas de processamento ou aceleradores específicos. Pesquisas atuais mostram que o *Field Programmable Gate Array* (FPGA) e a *Graphics Processing Unit* (GPU) apresentam bons resultados em diversas áreas de *Big Data*, como, por exemplo, o aprimoramento na taxa de transferência e consequentemente no desempenho em mecanismos de pesquisas na WEB (PUTNAM et al., 2014). Além de desempenho, o uso destes aceleradores (FPGA e GPU) podem proporcionar uma melhor eficiência energética como apresentado em (CHEN et al., 2014).

FPGAs e GPUs foram usados em várias áreas e ajudaram os pesquisadores a obter

bons resultados (SOUZA et al., 2018; MACIEL; SOUZA; FREITAS, 2019; CASTRO et al., 2019). Por exemplo, há melhoria na taxa de transferência de dados e, consequentemente, o desempenho nos mecanismos de pesquisa na Web (PUTNAM et al., 2014). Além das melhorias de desempenho, o uso desses aceleradores pode proporcionar melhor eficiência energética (CHEN et al., 2014; SOUZA et al., 2018).

Uma maneira de usar esses dispositivos de aceleração é em um contexto de computação heterogênea. Este campo de pesquisa refere-se ao uso de elementos de processamento de diferentes tipos em um sistema de computação, por exemplo: CPUs, GPUs e FPGAs. Sendo assim, em várias áreas (por exemplo, mineração de dados, inteligência artificial, aprendizado de máquina e FCA), o uso de sistemas heterogêneos é uma alternativa para melhorar o desempenho e a escalabilidade, além de melhorar a eficiência energética.

1.1 Problema

A característica binária no FCA vem de uma relação entre um conjunto de objetos e um conjunto de atributos. Quanto mais objetos e atributos tivermos, maior número de operações será realizado, consequentemente aumentando o tempo de processamento. Um dos principais desafios do FCA está diretamente relacionado ao tempo necessário para processar todos os registros de objetos em relação às combinações de atributos para extração de conceito formal.

Diversas técnicas que associam e processam os objetos e atributos são amplamente estudadas (RODRÍGUEZ-LORENZO et al., 2017). No entanto, há um problema combinatório, que é um fator crucial para o desempenho (NETO; ZÁRATE; SONG, 2018). Sendo assim, abordagens para lidar com a grande quantidade de dados de maneira oportuna é um grande desafio para a comunidade científica.

Vale ressaltar que, devido à necessidade de um excessivo recurso para processar algoritmos FCA, o consumo de energia não deve ser ignorado. Portanto, é necessário um melhor uso dos recursos computacionais para processar grandes quantidades de dados de maneira eficiente em termos de energia.

1.2 Objetivo

Como citado na Seção 1.1, recursos computacionais são fatores de grande importância para realizar o processamento de um grande volume de informações. Além dos recursos computacionais, para desempenho, é preciso estar atento ao consumo de energia. Dito isso, o objetivo deste trabalho é propor arquiteturas paralelas e heterogêneas para extração de conceitos formais utilizando um algoritmo paralelo de força bruta em

OpenCL. Esta proposta pretende manter alto desempenho com baixo consumo de energia. Para isso, foi escolhido o algoritmo de Força Bruta devido à sua baixa complexidade e altas características de dissociação. Também é implementado para ser executado nas plataformas heterogêneas CPU + FPGA e CPU + GPU. A linguagem *OpenCL* foi usada para programar a solução nessas plataformas híbridas.

1.2.1 Objetivos Específicos

A fim de construir e avaliar os protótipos deste trabalho, é necessário estabelecer os seguintes objetivos específicos, sendo eles:

- Desenvolver um algoritmo heterogêneo de Força Bruta paralelo, utilizando a linguagem *OpenCL* focado nas arquiteturas CPU + FPGA e CPU + GPU;
- Desenvolver um algoritmo homogêneo de Força Bruta paralelo utilizando *OpenMP*.
- Avaliar o desempenho e energia do algoritmo *Data-Peeler*.
- Avaliar o desempenho e energia entre as abordagens heterogêneas (citadas anteriormente) e homogênea (somente CPU);
- Apresentar uma alta escalabilidade para processar grandes conjuntos de dados, por exemplo, até 1 milhão de objetos, variando de 25 a 30 atributos.

1.3 Justificativa

FCA funciona com conjuntos de dados binários, conforme explicado na Seção 2.6. Assim, os algoritmos escritos no nível da máquina têm maior compatibilidade, devido à lógica binária que pode ser implementada no *hardware*. Portanto, é importante escolher o algoritmo FCA mais adequado para o desenvolvimento no nível do *hardware*. Algoritmos como *In-Close* e *Data-Peeler* tornam-se muito complexos quando se analisa o desenvolvimento de *hardware*. Além disso, esses algoritmos requerem um número maior de elementos lógicos quando comparados a uma implementação do algoritmo de Força Bruta. Por ser fracamente acoplado, este possui alto potencial para ser implementado em paralelo, tanto para arquiteturas homogêneas quanto heterogêneas.

Vale ressaltar também que, a área de paralelismo e computação heterogênea, tem ganhado cada vez mais espaço na comunidade científica, sendo utilizada em diversos cenários. Além disso, com o uso correto do conceito de computação heterogênea, é possível processar grandes conjuntos de dados com melhor desempenho e eficiência energética.

Outro ponto de destaque desta pesquisa é a escalabilidade dos protótipos com capacidade de processar grandes conjuntos de dados (por exemplo, 1 milhão de objetos com 25 atributos) que não é alcançada em alguns trabalhos atuais da área de FCA.

1.4 Organização da Dissertação

Esta dissertação está organizada de acordo com as seguintes seções: a Seção 2 apresenta a revisão da literatura. Já na Seção 3, são discutidos os trabalhos que possuem relação com esta pesquisa. A Seção 4 descreve a metodologia, e em seguida, a Seção 5 apresenta a proposta de arquitetura. Na Seção 6, é realizada a avaliação dos resultados obtidos no desenvolvimento da dissertação. Por fim, na Seção 7, são descritas as conclusões deste trabalho.

2 REVISÃO DA LITERATURA

Tempo de processamento e consumo de energia para a descoberta de conhecimento, *big data* e outras áreas estão intimamente relacionados à evolução das arquiteturas (NESHATPOUR et al., 2015). Nesse contexto, o campo da FCA tem um papel relevante e requer grande poder de processamento. Sendo assim, nesta seção, é apresentado alguns conceitos sobre arquitetura e FCA.

Com objetivo de facilitar o entendimento, separa-se esta seção da seguinte forma: a Seção 2.1 contextualiza o conceito de arquiteturas paralelas, em seguida a Seção 2.2 apresenta o conceito de computação heterogênea, a Seção 2.4 exibe o *chip* programável FPGA, já na Seção 2.5 são apontadas as características da GPU e por fim na Seção 2.6 é exibida a área de pesquisa de FCA.

2.1 Arquiteturas Paralelas

As primeiras arquiteturas de computadores realizavam uma quantidade muito baixa de operações lógicas, de forma a limitar o seu uso em diversos aspectos como uma simples calculadora, e com um grande espaço de tempo para sua execução. No entanto, com a evolução das arquiteturas, hoje é possível processar um volume de informações muito maior em um curto espaço de tempo.

O primeiro *chip* comercial foi lançado pela Intel em 1971 (FAGGIN et al., 1996), com uma frequência máxima de 740 kHz (740 mil ciclos por segundo), em que cada instrução levava um tempo de 8 ciclos para ser executado. Com o passar do tempo, novos estudos e novas arquiteturas foram criadas para melhorar o desempenho e processamento das instruções.

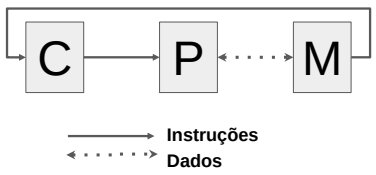
A necessidade de maior processamento das instruções com o menor intervalo de tempo possível, fez com que o aumento da frequência dos *chips* se tornasse uma solução conforme a Lei de Moore (NERI et al., 2003), estabelecendo que, quanto maior a quantidade de transistores, maior o desempenho da capacidade de integração em um circuito uma vez que facilita o aumento da frequência. Entretanto, o aumento da frequência esbarra nas limitações do semicondutor. Sendo assim, foi proposto o desenvolvimento de *chips* contendo mais de um núcleo de processamento com o objetivo de dividir o volume de dados entre eles, considerando que o aumento da frequência tornava-se inviável. Esses são conhecidos como *chips multi-core*, posteriormente *many-core*. Além disso, a principal

característica de um *chip many-core* está associada a quantidade elevada de núcleos (centenas), e a necessidade de uma rede de interconexão em *chip* mais robusta, a exemplo das *Network-on-Chip* (KUMAR et al., 2002).

Com o aumento dos núcleos de processamento em um *chip*, a classificação de execução de tarefas paralelas publicadas por Michael J. Flynn em 1966, torna-se adequada para arquiteturas em *chip multi-core* e *many-core*. Por se basear em um fluxo de dados e instruções durante a execução de um programa. Essa classificação é conhecida como Taxonomia de Flynn (FLYNN, 1966), que é dividida em quatro categorias. São elas:

a) *Single instruction, single data (SISD)*: O SISD é um tipo de arquitetura simples, composta por um processador que executa um fluxo de dados. Esses dados são operados 1 por vez, como é apresentado na Figura 1;

Figura 1 – SISD

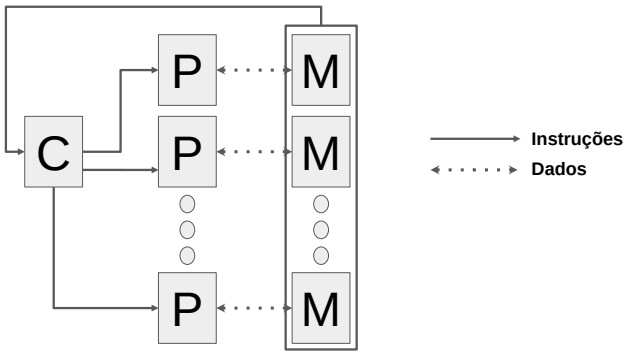


Fonte: (ROSE; NAVAUX, 2002)

b) *Single instruction, multiple data (SIMD)*: O SIMD tem um princípio semelhante ao SISD, executando um único dado por vez em uma unidade de processamento. Entretanto, ele possui mais de uma unidade com objetivo de executar dados em paralelo como apresentado na Figura 2. Essa arquitetura é bastante utilizada para operações em vetores e matrizes;

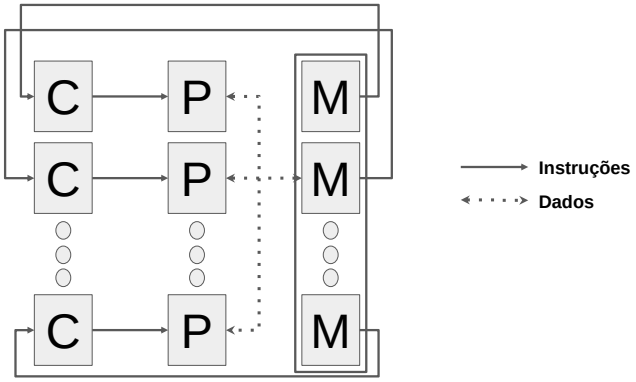
c) *Multiple instructions, single data (MISD)*: A arquitetura MISD é composta por mais de uma unidade de processamento diferentemente do SIMD como apresentado na Figura 3. Esta arquitetura recebe varias instruções distintas, no entanto, opera em apenas um único dado, sendo utilizada mais em aplicações de criptografia (para decodificar uma mensagem). A característica desta arquitetura faz com que o seu uso seja pouco aplicado na prática, já que na maioria dos casos, não é necessário repetir as instruções em um mesmo dado.

Figura 2 – SIMD



Fonte: (ROSE; NAVAUX, 2002)

Figura 3 – MISD

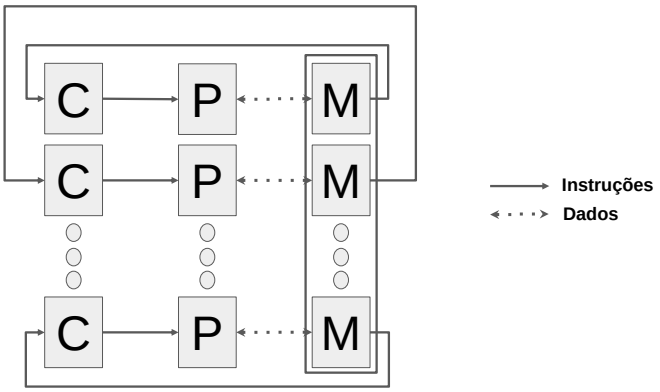


Fonte: (ROSE; NAVAUX, 2002)

d) *Multiple instructions, multiple data (MIMD)*: A arquitetura MIMD é um tipo de computação que executa as instruções em um conjunto de dados diferentes das outras unidades como é apresentado na Figura 4. Esta arquitetura possui vários processadores, cada um sendo controlado por uma unidade de controle. Ela recebe instruções distintas, podendo operar em diferentes dados com a possibilidade de trabalhar de forma síncrona ou assíncrona. Essa arquitetura é a mais utilizada atualmente em diversos *clusters*, computadores e processadores.

Como a característica da arquitetura MIMD possui maior flexibilidade em operações paralelas, foi utilizado como base para diversas propostas de arquiteturas e modelos de programação pela comunidade, como a computação heterogênea. É aceitável dizer que as arquiteturas heterogêneas proporcionam maiores oportunidades para pesquisas com foco em aumento de desempenho e baixo consumo de energia, em (ANDRADE; CRNKOVIC, 2018) é abordado 28 trabalhos utilizando computação heterogênea, de forma a comprovar o uso desta área. Na Seção 2.2 o tema é apresentado detalhadamente.

Figura 4 – MIMD



Fonte: (ROSE; NAVAUX, 2002)

2.2 Computação Heterogênea

As primeiras evoluções dos *hardware* visavam o aumento dos recursos computacionais, como a velocidade do *clock*, aumento de *cache*, dentre outros. No entanto, devido às limitações dos recursos, outras abordagens que se fizeram necessárias, tais como a computação heterogênea.

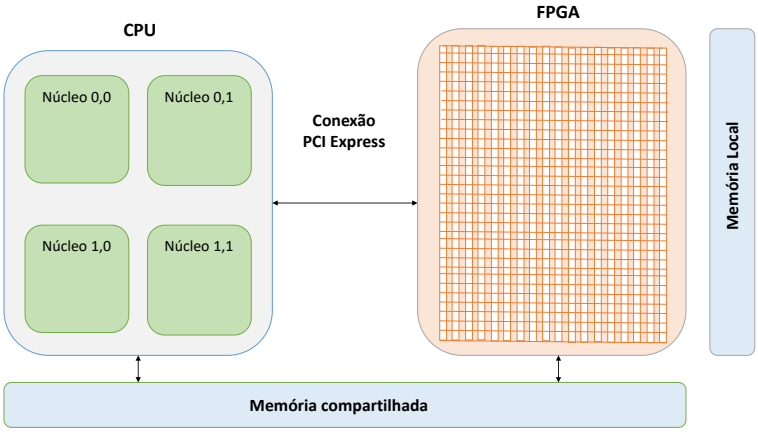
A expressão "computação heterogênea" representa as distintas formas de utilizar diferentes componentes dentro de um conjunto computacional para extrair o melhor desempenho de cada um (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Além disso, a combinação desses componentes tem como objetivo principal dividir as tarefas para conquistar um melhor desempenho da aplicação principal com uma menor quantidade de recursos.

A computação heterogênea também faz combinações com plataformas que possuem processadores do mesmo tipo, mas com capacidades diferentes. Por exemplo, um sistema que inclui mais de uma CPU com um número diferente de núcleos e/ou frequências de *clock*, pode ser chamado de heterogêneo. Em (KANG et al., 2018) são utilizados processadores com diversos núcleos em uma mesma plataforma, para ser aplicado métodos no *hardware* que aproveita cada núcleo de forma diferente em uma arquitetura heterogênea fortemente acoplada, ou seja, em um mesmo processador ou plataforma.

Assim como apresentado em (DONGARRA; LASTOVETSKY, 2009), há diversas pesquisas com diferentes plataformas heterogêneas como máquinas paralelas e sistemas distribuídos. Nas plataformas que são menos acopladas, os dispositivos que realizam processamentos específicos são denominados "aceleradores", que operam em conjunto as CPUs, como GPUs (*Graphics Processing Units*) e FPGAs (*Field-programmable Gate Arrays*).

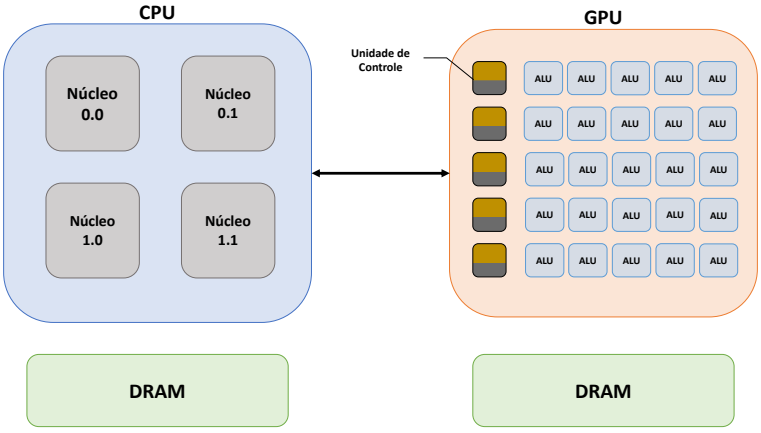
Como citado anteriormente, a computação heterogênea combina diferentes componentes/arquiteturas para um melhor desempenho, seja ele *software* e/ou *hardware*. Os sistemas heterogêneos são amplamente estudados como mostram no estado da arte (ANDRADE; CRNKOVIC, 2018). Além disso, o uso de aceleradores como FPGA ou GPU são alternativas que podem ser utilizadas em conjunto com as CPUs para realizar computação heterogênea, sendo estes dispositivos utilizados neste trabalho. Nas Figuras 5 e 6 são apresentados, respectivamente, exemplos das arquiteturas heterogêneas compostas por CPU+FPGA e CPU+GPU.

Figura 5 – Arquitetura Heterogênea CPU + FPGA



Fonte: Autoria própria

Figura 6 – Arquitetura Heterogênea CPU + GPU



Fonte: Autoria própria

Essas arquiteturas possuem distintas vantagens, como por exemplo:

- O FPGA possui a flexibilidade de reprogramar sua arquitetura (aproveitando o *hardware* da melhor maneira possível) e possui um baixo consumo de potência;

- A GPU realiza diversas operações lógicas em paralelo, tornando-a eficiente para operações recorrentes.

A fim de utilizar da melhor maneira possível os recursos dessas arquiteturas heterogêneas, é necessário ferramentas para o uso dos dispositivo em conjunto, como por exemplo VHDL (para FPGA), CUDA (paga GPU), *OpenCL* (multiplataforma), entre outras.

Neste trabalho utiliza-se o *OpenCL* para compilar e executar os algoritmos. Este será apresentado em detalhes na seção seguinte.

2.3 OpenCL

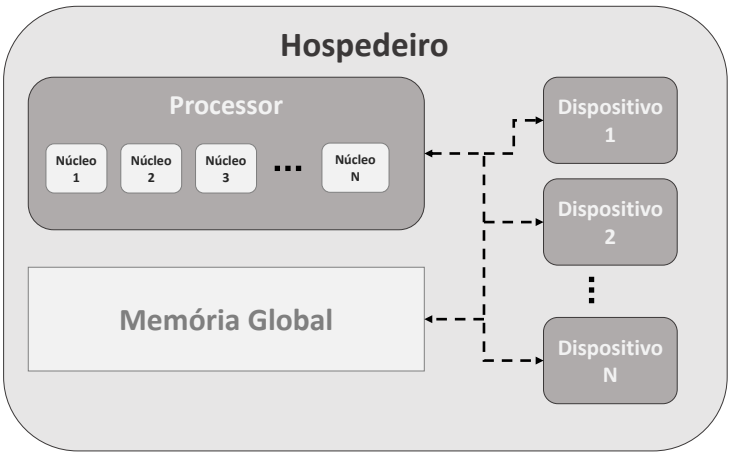
O *Open Computing Language* (*OpenCL*) é uma linguagem padronizada usada em programação de alto desempenho para ambientes de computação heterogêneos. A programação *OpenCL* é baseada em *kernels*, que são funções executadas em uma ou várias unidades ou dispositivos de processamento paralelo. Algumas das características desse idioma são:

- É uma linguagem multiplataforma, portanto, pode ser usada em vários sistemas operacionais ou *hardware*, como CPU, FPGA e GPU;
- Possui código portátil para as arquiteturas AMD, Intel e Nvidia;
- Suas especificações são baseadas nas linguagens C e $C++$;
- Funciona usando os conceitos de arquitetura SIMD e MIMD, assim como citado na Seção 2.1;
- Pode ser integrado a outras tecnologias.

A Figura 7 mostra uma visão geral do modelo de plataforma OpenCL. É composto por um hospedeiro (*host*), que se integra a um ou mais dispositivos (*devices*). Cada dispositivo possui vários grupos de trabalho (*Work Group*), que possuem um ou mais itens de trabalho (*Work Item*).

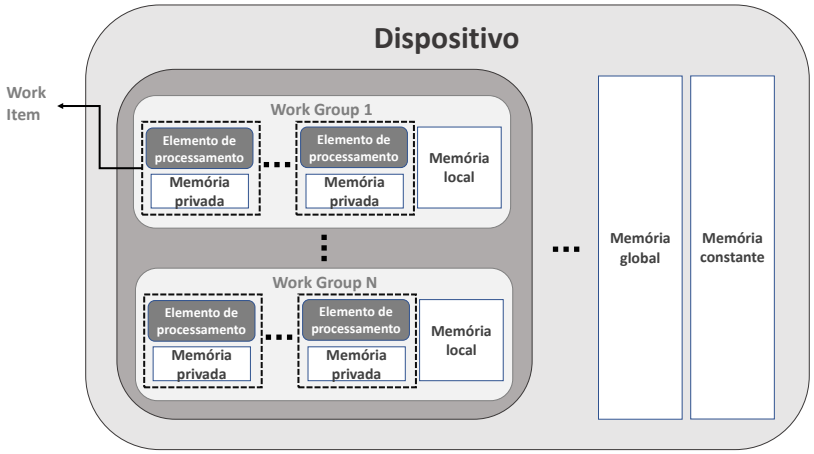
No modelo de plataforma do *OpenCL*, o *host* identifica e inicializa os dispositivos. Em seguida, ele transfere dados (envio e recebimento) e tarefas a serem executadas nos *devices*. A Figura 8 mostra uma visão geral da arquitetura de um *device*. Os dados podem ser organizados em *work group*, subdivididos em *work item*. As tarefas são implantadas em cada *work item* dentro do *work group*, para serem executadas em paralelo.

Figura 7 – Plataforma OpenCL (Host)



Fonte: (SILVEIRA; JR; CAVALHEIRO, 2010)

Figura 8 – Plataforma OpenCL (Device)



Fonte: (SILVEIRA; JR; CAVALHEIRO, 2010)

O número de *work group* e *work item* são configurados no *host*. Esses valores definem a maneira como a arquitetura do dispositivo distribuirá os dados a serem processados no *kernel* implantado. Essa configuração está diretamente ligada ao desempenho do algoritmo. Isso ajuda a fazer um melhor uso do *device* sem sobrecargas ou até mesmo mau uso dos recursos disponíveis.

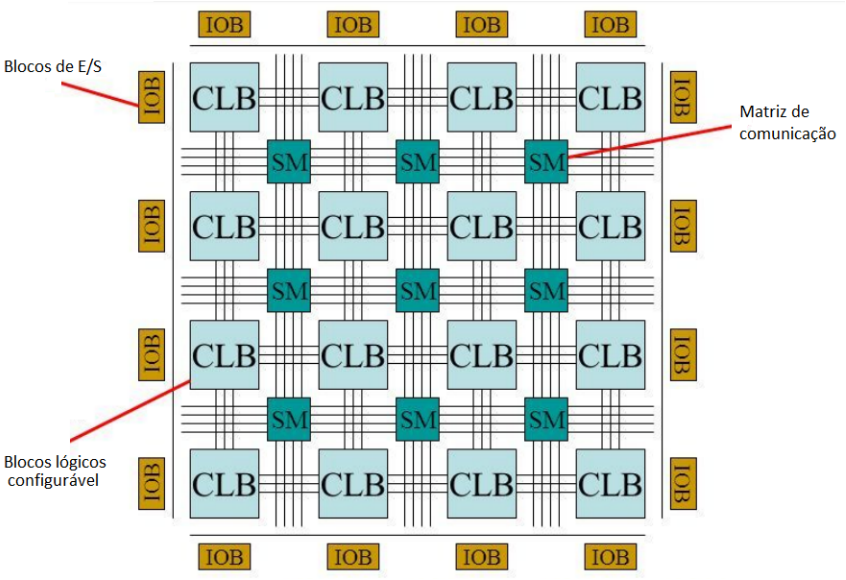
Considerando o uso dessa plataforma para desenvolvimento em arquiteturas heterogêneas, nas Seções 2.4 e 2.5 são apresentados em maior detalhe os dispositivos utilizados neste trabalho.

2.4 Field Programmable Gate Array (FPGA)

O *Field Programmable Gate Array* (FPGA) é um *hardware* que possui diversos componentes lógicos desconectados. Estes componentes necessitam ser pré-configurados, criando uma lógica de circuitos em um único *chip* para executar ações específicas de acordo com sua configuração. Embora sua arquitetura seja mais simplificada do que a dos demais aceleradores (como a GPU), ele permite ser configurado para executar várias instruções em paralelo (em nível de *hardware*), como adições e multiplicações, e também contendo um baixo consumo de energia.

A Figura 9 mostra a estrutura simplificada do FPGA. Esta estrutura é composta por 3 componentes, sendo eles: blocos de entrada e saída, blocos lógicos configuráveis e matriz de comunicação. Esses componentes são necessários para configurar de forma inteligente os elementos lógicos e compor um circuito.

Figura 9 – Arquitetura FPGA



Fonte: Autoria própria

Assim, o FPGA consiste num conjunto de blocos lógicos reconfiguráveis, além de blocos de processamento digital de sinais e opcionalmente um ou mais núcleos de CPU de propósito geral, todos conectados por uma interconexão. Esta interconexão é reconfigurável e pode ser utilizada para adaptar as aplicações aos FPGAs. Quando programado, os FPGAs funcionam como os circuitos integrados feitos para aplicações específicas (BRODTKORB et al., 2010).

A fim de reprogramar os circuitos integrados do FPGA para que execute as aplicações específicas, é necessário realizar a descrição do *hardware* utilizando uma linguagem de programação. Sendo assim, as principais linguagens de programação para descrição de

hardware são: VHDL e *Verilog*. No entanto, para trabalhos que envolvam computação heterogênea, linguagens como *OpenCL* e *OpenACC* também podem ser utilizadas.

A flexibilidade de reprogramar os circuitos integrados do FPGA permite a criação de arquiteturas dedicadas/específicas para determinados algoritmos. Sendo assim, o FPGA é uma boa escolha para a computação heterogênea, isto é, o FPGA pode trabalhar como um acelerador para a CPU com objetivo de realizar operações específicas para um grande volume de informações.

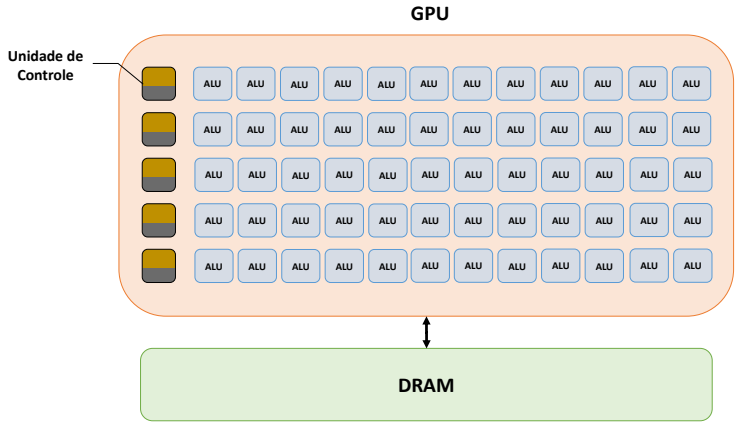
A próxima seção apresenta uma breve explicação sobre a GPU, sendo ela, um outro acelerador que também pode ser utilizado em arquiteturas heterogêneas.

2.5 Graphics Processing Unit (GPU)

O *Graphics Processing Unit* (GPU) é um microprocessador projetado para realizar operações gráficas, em que, na maioria dos casos, são usados para um melhor desempenho de jogos, realidade virtual, renderização 3D, simulações de sistemas complexos, entre outras. No entanto, devido a sua arquitetura com centenas de núcleos de processamento e consequentemente sua característica paralela, torna-se adequada para processar um alto volume de informações recorrentes ao mesmo tempo.

Como apresentado na Figura 10, as GPUs são compostas por diversas unidades lógicas aritméticas (ULAs) que são controladas por uma Unidade de Controle e possuem seu próprio *cache* cada. Além disso, as GPUs possuem uma memória DRAM compartilhada, sendo esta conectada com as demais unidades para compartilhamento das informações e resultados.

Figura 10 – Arquitetura GPU



Fonte: Autoria própria

Além das utilizações citadas anteriormente para as GPUs, devido as suas caracte-

rísticas paralelas, esse dispositivo torna-se alvo para pesquisas de processamento com alto volume de informações como (GUTIÉRREZ et al., 2017) que mostra o uso de GPU para mineração de dados.

Na Seção 2.6 é apresentada a área de pesquisa para extração de conceito formal (FCA). Esta área de pesquisa exige grande processamento de informações, diante disso, a computação heterogênea pode ser um caminho para obter um melhor desempenho.

2.6 Análise Formal de Conceitos

Análise Formal de Conceitos, do inglês, *Formal Concept Analysis* (FCA) é um ramo da matemática denominada teoria de reticulados iniciado por (DAVEY; PRIESTLEY, 1990), e composta por conjuntos parcialmente ordenados, que possuem relações binárias. Essa teoria foi profundamente estudada em sua forma geral de estrutura por (GRÄTZER, 2002), e discutida em mais detalhe por (WILLE, 2009) em uma forma hierárquica reestruturada.

Os fundamentos matemáticos para a área de FCA são apresentadas em (GANTER; WILLE, 2012). De um modo geral, esse ramo de pesquisa se baseia em três definições, que também dão seu nome:

- *Analysis*: refere-se ao fundamento matemático da geração de conceitos através da análise;
- *Concept*: consiste em conjuntos nos quais os dados são estruturados, para interpretação humana;
- *Formal*: refere-se à observação e manipulação dos dados de destino.

A FCA extrai conceitos, ou conhecimento, que são gerados a partir de um conjunto de dados e suas dependências (implicações). Sendo o conjunto de dados - com objetos, atributos e incidências - formando um contexto formal. A seguir, nas Seções 2.6.1 e 2.6.2, são explicadas o significado desses termos e o processo de extração.

2.6.1 Contexto Formal

Um contexto formal é composto por dois elementos, objetos e atributos, e uma relação binária denominada incidência. Portanto, esse contexto pode ser representado pela notação $C = (O,\ A,\ R)$, no qual O e A representam respectivamente objetos e atributos e $R \subseteq O \times A$ uma relação binária chamada incidência. Todo elemento da

incidência tem a notação oRa. Em outras palavras, um relacionamento entre objeto e atributo é lido como "o objeto o tem o atributo a".

A tabela de incidência é uma maneira adequada de representar um contexto formal (Tabela 1). Linhas representam os objetos m e colunas os atributos n. Quando um objeto tem o atributo, uma incidência é representada por um símbolo (por exemplo, "X").

Tabela 1 – Tabela de incidência

	A1	**A2**	**...**	**Am**
O1	x	x	...	
O2	x		...	x
...	...	...	...	...
On		x	...	x

Resumindo, contextos formais são conjuntos com objetos que possuem atributos ou não, ou seja, o relacionamento entre esses dois elementos é verdadeiro ou falso. Vale ressaltar que, em conjuntos de dados comuns, os atributos podem ter vários valores, portanto, é necessário definir um contexto formal de múltiplos valores. Ou seja, se um atributo A possui dois valores, é necessário dividi-lo em outros dois ($A1$ e $A2$); consequentemente, é possível realizar a análise de forma binária (verdadeira ou falsa).

2.6.2 Conceitos Formais

Um contexto formal define conceitos formais com base em alguns operadores de derivação. Para cada subconjunto $E \subseteq O$ e $I \subseteq A$, seus conjuntos derivados podem ser determinados por meio de operações de derivação:

$$E' = \{a \in A \mid \forall o \; E \; oRa\}$$

$$I'' = \{o \in O \mid \forall a \; I \; oRa\}$$

Após o processo de derivação, para cada conjunto de atributos (combinações) que possuem um conjunto de objetos, se esse conjunto de objetos tiver o mesmo conjunto inicial de atributos idênticos, teremos um conceito formal. Por exemplo, veja a Figura 11. O conjunto de atributos $\{A1, A3\}$ foi derivado, produzindo o conjunto de objetos $\{O1, O3\}$. Essa relação é um conceito formal, porque o processo de derivação do conjunto $\{O1, O3\}$ encontra exatamente o mesmo conjunto inicial $\{A1, A3\}$. Por outro lado, o conjunto $\{A1\}$, quando derivado, produz o conjunto $\{O1, O3\}$. No entanto, esse conjunto de objetos não produz $\{A1\}$ quando derivado, portanto, essa relação não é um conceito formal.

Figura 11 – Conceito Formal

	A1	A2	A3	A4
O1	X	X	X	
O2		X	X	
O3	X		X	
O4		X		X

Conceito	
a)	A1,A3 => O1, O3
b)	O1, O3 => A1,A3

Não é Conceito	
a)	A1 => O1,O3
b)	O1,O3 => A1, A3

Fonte: Autoria própria

2.6.3 Algoritmos para Extração de Conceitos

Atualmente, existem vários algoritmos que podem ser usados para encontrar conceitos formais em contextos formais. Alguns exemplos são *In-Close* (e suas variações) (ANDREWS; ORPHANIDES, 2010), *Data-Peeler* (CERF et al., 2008, 2009, 2013) e uma abordagem de Força Bruta.

In-Close e *Data-Peeler* são algoritmos que usam abordagens para evitar comparações desnecessárias, a fim de diminuir o tempo de processamento em grandes contextos. No entanto, devido à complexidade das heurísticas que eles usam, a maioria das implementações desses algoritmos é feita de maneira sequencial. Portanto, eles não podem fazer o melhor uso das arquiteturas paralelas modernas com muitos núcleos de processamento.

Por outro lado, uma abordagem de Força Bruta é mais simples e mais fácil de manipular do que os algoritmos mencionados. Apesar disso, sua principal desvantagem é a necessidade de comparar todos os conjuntos possíveis de derivações. É isso que prolonga o tempo de processamento nas execuções sequenciais. Mesmo assim, o algoritmo de Força Bruta tem uma complexidade muito baixa. Assim, sua estrutura é altamente compatível com o paralelismo de instruções, permitindo um ótimo desempenho em arquiteturas *multi-core*.

2.6.4 Força Bruta

Como já mencionado na seção anterior, *In-Close* e *Data-Peeler* são algoritmos com um grau de complexidade maior, por exemplo, dependência de dados, o que dificulta sua implementação com métodos paralelos. Ao contrário, uma abordagem de Força

Bruta tem baixa complexidade e alto desacoplamento. Essas são características vantajosas para implementá-lo para execução em *hardware* paralelo ou sistemas heterogêneos. Dessa forma, foi selecionado para implementar um algoritmo paralelo de Força Bruta para uso em sistemas heterogêneos.

Referindo-se às Seções 2.6.1 e 2.6.2, recordando o processo para gerar um conceito formal. Primeiro, é enviado um grupo de atributos para uma função de derivação, capaz de gerar uma combinação de atributos. Por exemplo, dado um conjunto de atributos $\{a, b, c\}$, as seguintes combinações serão geradas: $\{\varnothing\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$ e $\{a, b, c\}$. Cada conjunto de atributos das combinações implica que um conjunto de objetos o contenha, como um "conjunto inicial de atributos". O conjunto de objetos, obtido a partir da primeira derivação, também é sujeito ao processo de derivação. Implica um novo conjunto de atributos, que forma um conceito formal se for igual ao inicial.

Para manter um alto desacoplamento, o algoritmo de Força Bruta tem duas etapas principais. São elas, (i) gerar combinações e (ii) análise de conceito.

Algoritmo 1: Extração de Conceitos

```
1 Contexto ← LerContextoDoArquivo()
2 NumAtributos ← QuantidadeAtributos()
3 ResultadoConceitos ← 0
4 Combinacoes ← GeradorCombinacoes()
5 para (i = 1 → Combinacoes.Length) faça
6 |   eConceito ← AnaliseDeConceito(Combinacoes[i], Contexto);
7 |   se eConceito então
8 |   |   ResultadoConceitos + +;
9 retorna ResultadoConceitos;
```

O Algoritmo 1 apresenta o algoritmo de Força Bruta, com estas etapas. Primeiro, a entrada *NumAtributos* é necessária para gerar todas as combinações possíveis de atributos. A seguir, a entrada *Contexto* corresponde aos objetos e atributos a serem derivados. A função *GeradorCombinacoes* na linha 4 é executada para criar todas as combinações de atributos. Em seguida, o conjunto de combinações produzidas é percorrido pelo *loop* na linha 5. Cada combinação é analisada no contexto e as respectivas derivações são feitas (atributos aos objetos e vice-versa). Esse processo ocorre na linha 6. Após a análise de cada combinação, se ela produzir um conceito formal, a variável *ResultadoConceitos* é incrementada. Finalmente, quando todas as combinações são avaliadas, o número de conceitos formais encontrados é retornado e a lista de conceitos gerados é armazenada pelo algoritmo.

3 TRABALHOS RELACIONADOS

A extração de conceitos tem uma complexidade combinatória e é comum encontrar contextos formais com alta dimensionalidade (muitos atributos e objetos). A literatura apresenta vários algoritmos criados para extrair conceitos. Eles tentaram principalmente possibilitar o processamento de contextos com alta dimensionalidade. Ou seja, o principal objetivo no trabalho relacionado é extrair conhecimento de conjuntos de dados cada vez maiores e complexos.

Em (ANDREWS, 2011) é apresentado o algoritmo *In-Close2*, sendo uma variante aprimorada do método *Close-by-One* (CbO). Ele é um algoritmo iterativo, que depende do resultado anterior para executar o próximo, a fim de reduzir a complexidade do contexto formal. No trabalho citado, o algoritmo é comparado com outra variante do CbO: *Fast Close-by-One* (FCbO). O *In-Close2* mostra um desempenho melhor que o FCbO na maioria dos casos, sendo altamente recomendado para vários casos avaliados.

Em outro trabalho, algoritmos como CbO, FCbO, *In-Close*, *In-Close2* e *In-Close3*, mostram que podem ser usados em aplicativos com *Galois Connections* (ANDREWS, 2015). *Conexões Galois* são pontos fixos, isto é, uma relação específica entre dois conjuntos parcialmente ordenados. Eles formam padrões em dados relacionais binários, permitindo que sejam usados na área da FCA. Para esse tipo de aplicação, os autores afirmaram que os algoritmos baseados em CbO apresentam melhores resultados.

Em (KODAGODA; ANDREWS; PULASINGHE, 2017) é retratado em detalhes o algoritmo *In-Close3*. Este apresenta uma estratégia de paralelização para extração de conceito formal e faz uso da estrutura *OpenMP*. Este algoritmo divide o contexto em sub-árvores para realizar o paralelismo em diversas *threads*. No entanto, assim como dito pelo autor, este trabalho necessita de um estudo e testes mais aprofundados para exibir o desempenho real da abordagem, uma vez que os resultados apresentados foram apenas simulados.

Outro algoritmo que extrai conceitos é o *Data-Peeler* (CERF et al., 2008, 2009, 2013). Este é considerado o algoritmo de última geração para esta tarefa, apresentando também bom desempenho. O *Data-Peeler* implementa métodos que evitam comparações desnecessárias no contexto formal. Esse algoritmo pode ser usado em várias aplicações para identificar padrões em *clusters* (HENRIQUES; MADEIRA, 2018), para otimizar comparações por meio da remoção da estrutura de dados, para evitar operações recorrentes e consequentemente para reduzir o tempo de processamento.

A busca de algoritmos para FCA que possam processar contextos de alta dimensionalidade tem sido o foco principal da pesquisa nos últimos anos. Por exemplo, o algoritmo *ImplicPBDD* (SANTOS et al., 2018) é uma variante do algoritmo *PropIm*, mas usa uma estrutura de diagrama de decisão binária (BDD). Tem como objetivo lidar com a alta dimensionalidade, simplificando a estrutura de dados do contexto formal com um BDD. Os autores mostraram que o *ImplicPBDD* pode apresentar desempenho até 80% maior que o seu antecessor. A avaliação foi realizada sobre conjuntos de dados sintéticos, gerados com diversos tamanhos e densidades. A densidade em FCA está relacionada a quantas relações de objetos e atributos no contexto formal existem, correspondendo ao nível de incidência de atributos em cada objeto.

Contudo, há trabalhos que tratam problemas com alta dimensionalidade no número de objetos. O trabalho apresentado em (MORAES et al., 2016), apresenta uma abordagem que utiliza computação paralela como um meio de reduzir o tempo de análise para cenários contendo contextos de entrada com alta densidade e dimensionalidade. De acordo com o autor, os experimentos mostram uma redução de aproximadamente 75% no tempo de execução utilizando uma abordagem paralela baseada no algoritmo *NextClosure*.

As pesquisas em FCA são importantes em várias aplicações, como mineração de dados, robótica e medicina. A comunidade científica realizou estudos combinando os fundamentos de FCA e um *Fuzzy Logic Controller* (FLC) (NEELIMA; SARMA, 2019), para obter respostas de uma ação humana. Essa abordagem analisa dados que têm relações entre o conjunto de objetos e atributos anteriores das ideias do pensamento humano. Ele gera conceitos a serem aplicados no FLC, construindo regras *Fuzzy* para, posteriormente, obter respostas a uma ação humana.

Outra maneira de aplicar os fundamentos de FCA, está em identificar estruturas conceituais nas redes sociais (NETO et al., 2018). Neste trabalho, modelos computacionais foram baseados em implicações que representam e analisam subestruturas de redes sociais. Os autores puderam encontrar padrões de acesso em tais redes, concluindo que eles representam comportamentos específicos.

Como foi apontado, há uma grande necessidade de recursos computacionais para processar grandes volumes de informações. A pesquisa apresentada em (ANDRADE; CRNKOVIC, 2018), mostrou que o uso de arquiteturas heterogêneas com aceleradores (FPGA e GPU) provou ser bastante promissor em vários aspectos relacionados aos grande volume da área de *big data*. Alguns trabalhos realizados também usaram aceleradores personalizados para uma estrutura do *Hadoop MapReduce* (NESHATPOUR et al., 2015).

Vários estudos mostram a importância de extrair um conceito formal, bem como as vantagens que arquiteturas de processamento específicas oferecem. Contudo, trabalhos que usam computação heterogênea, tem apresentado bons resultados para processar gran-

des conjuntos de dados. Além disso, esta dissertação consiste na extensão do trabalho publicado em (MACIEL et al., 2019), com as seguintes contribuições:

- Um algoritmo heterogêneo de força bruta OpenCL focado nas arquiteturas CPU + FPGA e CPU + GPU para extração de conceitos;
- Avaliação de desempenho e energia entre as abordagens deste trabalho com um algoritmo mais inteligente;
- Alta escalabilidade para processar grandes conjuntos de dados, por exemplo, até 1 milhão de objetos, variando de 25 a 30 atributos.

Considerando os trabalhos citados anteriormente, o objetivo é analisar os ganhos que são possíveis de obter utilizando os fundamentos de FCA ao aplicar computação heterogênea. Vale ressaltar que, a computação heterogênea em FCA é uma área de pesquisa que ainda não é muito explorada. Mesmo que algumas pesquisas tenham sido iniciadas com abordagens paralelas. Para obtenção de um melhor desempenho na extração de conceitos (como mostra em (KODAGODA; ANDREWS; PULASINGHE, 2017)), ainda se faz necessário aprofundar nessas áreas, já que a complexidade de paralelizar um algoritmo recursivo não é trivial. Além disso, o uso de algoritmo com altas características de desacoplamento (isto é, o algoritmo de Força Bruta), é necessário para um melhor uso de plataformas heterogêneas. Na próxima seção, é apresentada a metodologia proposta para o projeto de uma arquitetura heterogênea capaz de extrair conceitos formais de contextos formais.

4 METODOLOGIA

A fim de propor arquiteturas paralelas e heterogêneas(CPU+FPGA e CPU+GPU) e desenvolver um algoritmo de Força Bruta paralelo para extração de conceito formal, é estabelecida uma metodologia apresentando os materiais e método aplicados para desenvolvimento dos protótipos.

Sendo assim, a Seção 4.1 mostra os materiais que são necessários no desenvolvimento. Já na Seção 4.2 são apresentadas etapas para desenvolvimento do projeto. Por fim na Seção 4.3 é detalhado como é realizada a avaliação da arquitetura.

4.1 Materiais

O algoritmo de Força Bruta paralelo é desenvolvido com objetivo de executar em uma plataforma composta por CPU e FPGA/GPU. Sendo assim, é importante lembrar que essa plataforma consiste em um *host* e um *device*, em que o *host* corresponde a CPU, já o *device* equivale ao FPGA ou GPU. Logo, a fim de atingir o objetivo, na Seção 4.1.1 é apresentado os materiais para a criação da arquitetura homogênea, e na sequência na Seção 4.1.2 é mostrado os materiais da arquitetura heterogênea.

4.1.1 Arquitetura Homogênea

Na arquitetura Homogênea é desenvolvido o algoritmo (para o *host*) do Força Bruta. Além disso, também é executado o algoritmo *Data-Peeler*. Ambos possui objetivo de extração de conceitos. Para isso, são utilizados os seguintes materiais:

- Linguagem C e *OpenMP*;
- GCC para compilação;
- Base de dados sintética (gerada pelo *software* SCGaz);
- Máquina com 2 processadores Intel ® Xeon ® E5-2620 v2 @ 2.10GHz com 64GB RAM e 24 *threads* (cada processador possui 6 núcleos e 12 *threads*);
- Máquina com 2 processadores Intel ® Xeon ® E5-2699 v4 @ 2.20GHz com 64GB RAM e 88 *threads* (cada processador possui 22 núcleos e 44 *threads*).

4.1.2 Arquitetura Heterogênea

Na arquitetura heterogênea, o algoritmo implementado para o *host* gera as combinações dos atributos com objetivo de enviar ao *device*, já no *device* o algoritmo projetado corresponde ao Força Bruta para extração de conceito de cada combinação recebida do *host*. Para isso, foi necessário utilizar os seguintes materiais:

- Linguagem *C* e *OpenCL*;
- GCC para compilação;
- Máquina com 2 processadores Intel ® Xeon ® E5-2620 v2 @ 2.10GHz com 64GB RAM e 24 *threads* (cada processador possui 6 núcleos e 12 *threads*);
- Máquina com 2 processadores Intel ® Xeon ® E5-2699 v4 @ 2.20GHz com 64GB RAM e 88 *threads* (cada processador possui 22 núcleos e 44 *threads*);
- FPGA Intel *Arria 10* com até 1.150k de elementos lógicos;
- GPU *Nvidia Tesla* K20m com 2496 núcleos;
- Base de dados sintética (gerada pelo *software* SCGaz);
- *Quartus* 16.0 - Ferramenta de compilação do FPGA;
- CUDA *Toolkit* 10 - Ferramenta de compilação da GPU;
- *Powertop*;
- *Model Sim* - Ferramenta para simulação da arquitetura em FPGA.

4.2 Método

No desenvolvimento das arquiteturas homogêneas e heterogêneas foram necessários realizar os seguintes passos: pesquisar projetos de extração de conceitos, pesquisar conceitos de computação heterogênea, projetar o algoritmo de Força Bruta apenas no *host*, projetar o algoritmo que gera as combinações no *host* e o de Força Bruta no *device*, e por fim comparar a eficiência de ambos algoritmos junto com algoritmos mais recentes do estado da arte.

Um dos algoritmos mais utilizados para extração de conceito formal é o *Data-Peeler*, superando diversos algoritmos em desempenho e eficiência, como apresentado em (CERF et al., 2008). Devido aos seus resultados em performance, este algoritmo é usado como base de comparação neste trabalho.

Nas pesquisas de projetos de extração de conceitos, o maior gargalo que a comunidade científica encontra é o tempo para processar um grande volume de informações com alta dimensionalidade, tornando difícil a escalabilidade das abordagens tomadas. Sendo assim, uma possível solução encontrada (durante a pesquisa) para este gargalo, foi a computação heterogênea que até a data de conclusão desse trabalho não é muito explorada pela comunidade dessa área de pesquisa.

Após as pesquisas citadas acima, foram projetadas duas arquiteturas heterogêneas e uma homogênea baseadas no algoritmo de Força Bruta para extração de conceito formal, sendo elas:

- Arquitetura Homogênea: desenvolvida exclusivamente para o *host* em uma linguagem de programação *C* e *OpenMP*, e executada em uma máquina com 24 e 88 *threads* para processamento. Esta arquitetura é composta pelo algoritmo que gera todas as combinações de atributos junto com o algoritmo de comparação e extração dos conceitos formais da base de dados;

- Arquitetura Heterogênea: diferentemente da arquitetura homogênea, esta é desenvolvida em uma linguagem de programação *C* e *OpenCL* em que o *host* gera as combinações de atributos e configura o *device* para processar as combinações no FPGA ou GPU. O algoritmo descrito para o FPGA é o mesmo que o da GPU, sendo necessário ser alterado nas configurações do *host*.

O algoritmo de Força Bruta, como apresentado na Seção 2.6.4, possui baixa complexidade e alto desacoplamento, possibilitando sua execução paralela. Dito isso, este trabalho utiliza a abordagem de Força Bruta para desenvolver um algoritmo específico para execução em arquiteturas com múltiplos núcleos, podendo ser homogêneo (somente CPU) ou heterogêneo (CPU+FPGA ou CPU+GPU). Sendo assim, a utilização do *OpenMP* no *host* possibilita o paralelismo sem alteração nos resultados. Já nos *devices*, o conceito de *Work Group* apresentado na Seção 2.3 viabiliza a execução de diversas combinações em paralelo nos dispositivos, provendo um aproveitamento superior de recursos dos *devices* e mantendo o mesmo resultado final, porém com desempenho melhor em diversos casos.

A forma de análise das diversas arquiteturas desenvolvidas para este trabalho são apresentadas na Seção 4.3. A descrição dessas arquiteturas foram necessárias para as comparações entre si, juntamente com os projetos atuais do estado da arte. Estas comparações são melhor apresentadas no Capítulo 6. Já no Capítulo 5 é apresentado um melhor esclarecimento das propostas heterogêneas.

4.3 Avaliações

Para comprovar a eficiência do desempenho do algoritmo de Força Bruta proposto neste dissertação, foi utilizado o algoritmo *Data-Peeler* para comparação. Diante disso, foi necessário realizar as seguintes avaliações:

- O algoritmo de Força Bruta (baseado em *OpenMP*), foi executado na arquitetura homogênea, utilizando os processadores da Intel com 24 e 88 *threads*;
- O algoritmo de Força Bruta (baseado em *OpenCL*), foi executado na arquitetura heterogênea, utilizando os processadores da Intel (como *host*) e a GPU ou FPGA (como *device*);
- O *Data-Peeler* é um código-fonte aberto e, para sua execução, é necessário apenas adequar o conjunto de dados (contexto) para sua entrada. Foi executado na máquina com 88 *threads*. No entanto, devido à sua característica sequencial, ele não explorou mais de um núcleo de processamento.

A fim de avaliar o desempenho dos algoritmos, os contextos foram gerados com três densidades diferentes: 30%, 50% e 70%. Para cada valor de densidade, foi produzido sete contextos diferentes, variando de 1.000 a 1.000.000 objetos, em cada atributo. Além disso, a configuração que apresenta os melhores resultados foi selecionada para avaliar o desempenho do algoritmo quando o número de atributos é alterado. Também foi realizado variação de 25 a 30 atributos, com os três valores de densidade. A execução variando os atributos, foi usada para comparar as implementações do Força Bruta com o algoritmo *Data-Peeler*.

Em todos os testes, os resultados utilizam a média aritmética de 10 execuções. Dito isso, foi possível verificar que mais de 90% dos testes estão dentro do desvio padrão calculado. Em seguida, é avaliado o consumo de energia para cada proposta da seguinte forma:

- No algoritmo paralelo para CPU é usado a ferramenta *Powertop* para obter o consumo de energia;
- Já para a GPU, é utilizado os recursos do *CUDA Toolkit 10* para apresentar o consumo energético;
- Para o FPGA são utilizados os recursos da ferramenta *ModelSim* para obter o gasto de energia.

Por fim, é avaliada a eficiência energética (operações por *Watt*), sendo apresentadas no Capítulo 6.

5 PROPOSTA DE ARQUITETURAS PARALELAS PARA EXTRAÇÃO DE CONCEITOS

Para esta dissertação foi pensado um projeto homogêneo e heterogêneo baseado nas linguagens C e $OpenCL$, e a biblioteca $OpenMP$. Sendo assim, na Seção 5.1 é apresentado o funcionamento da proposta do projeto homogêneo, já na Seção 5.2 é apresentada a proposta do projeto heterogêneo.

5.1 Projeto Homogêneo

Diferentemente da arquitetura heterogênea, a arquitetura homogênea faz todo o processo no mesmo dispositivo (*host*) de forma que as combinações e análise de conceitos são realizadas apenas pela CPU, seguindo os passos apresentados na Seção 2.6.4. O algoritmo para esta proposta executa as seguintes demandas:

1. Gera todas as combinações de atributos, que são armazenadas na memória do *host*;
2. As tarefas de análise de conceitos são distribuídas por todas as *threads* disponíveis, para cada combinação.
3. Cada *thread* executa a derivação de cada combinação de atributos em paralelo.
4. A *thread* principal controla o número de conceitos formais obtidos em uma variável de retorno, até que todas as combinações sejam processadas.

Na seção seguinte é apresentado em detalhes o funcionamento do algoritmo para análise de conceitos, sendo o mesmo processo para o algoritmo homogêneo, diferenciando apenas a parte de *device*.

5.2 Projeto Heterogêneo

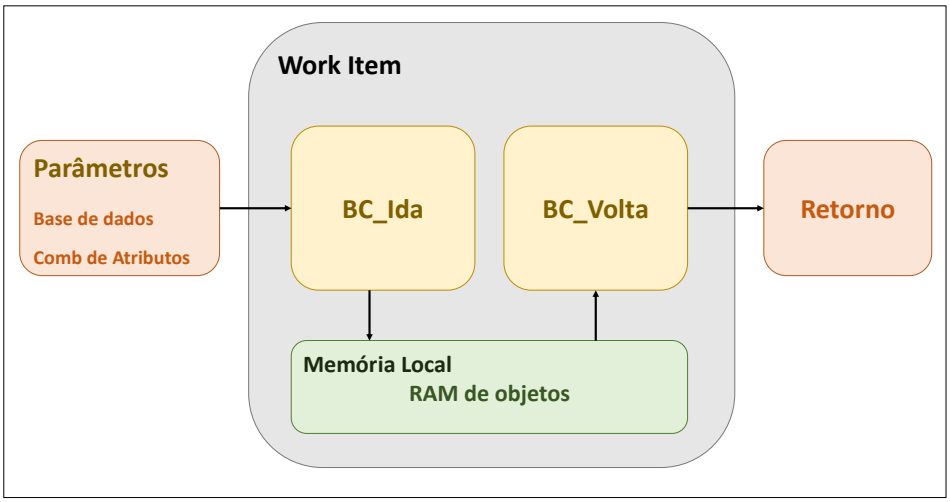
A execução do algoritmo de Força Bruta em uma arquitetura heterogênea, requer o uso do *host* e de um ou mais *devices*. O primeiro é geralmente uma CPU, enquanto o último pode ser um FPGA (Figura 5) ou uma GPU (Figura 6).

Para usar a arquitetura híbrida, foi criado um algoritmo para o *device*, chamado *kernel*. Este *kernel* é descrito usando o *OpenCL* em um arquivo com extensão *.cl*, que depende das interações com o *host* para executar suas ações. Portanto, é utilizado trechos

de código em C para implementar tarefas a serem executadas no *host*. Essas tarefas configuram os *devices*, *buffers* de memória e parâmetros a serem usados no *kernel*. Além disso, é escrito algumas instruções para criar o *kernel* no *host* e implantá-lo no *device*.

O algoritmo é executado em *Work Item*, como mostra a Figura 12. A arquitetura contém dois parâmetros de entrada importantes: "Base de dados", que é o contexto formal; e a "Comb de atributos". Ambas as entradas são armazenadas em uma memória global, compartilhada entre todas as unidades de computação, como mostra a Figura 13. Essas figuras mostram a arquitetura do *kernel*, que é montada após a compilação do algoritmo criado no *OpenCL*, seguindo as complexidades de cada dispositivo.

Figura 12 – Arquitetura Work Item



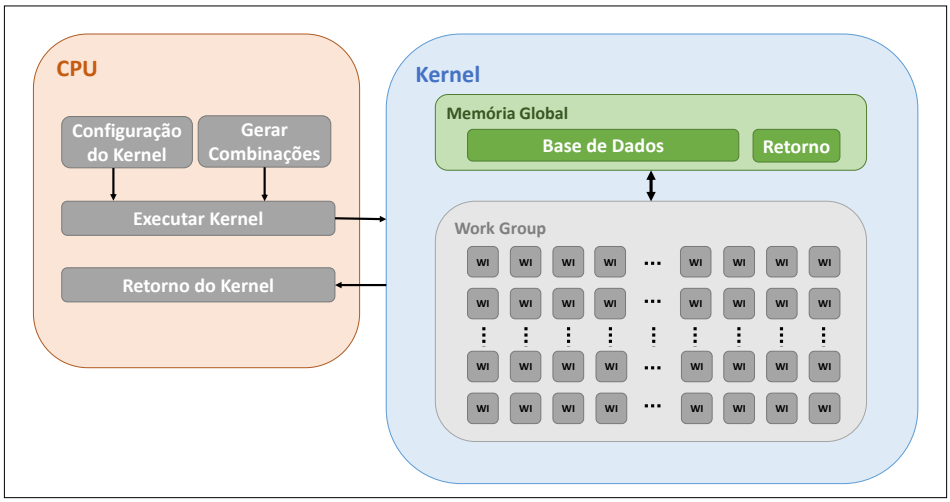
Fonte: Autoria própria

O parâmetro "Base de dados"permanece inalterado durante a execução do algoritmo. Como ele é armazenado na memória global, todos os itens de trabalho têm acesso a ele. As combinações de atributos são geradas pelo *host*, armazenadas em uma matriz A e enviadas para a memória global. Cada item de trabalho usa uma combinação diferente de atributos. Portanto, eles recebem uma combinação de atributos $A[i]$ onde i corresponde ao índice de combinação.

O algoritmo de Força Bruta que ocorre em cada *Work Item* é composto de três blocos. O primeiro, BC_Ida, executa o processamento inicial dos conceitos. Em seguida, o resultado de BC_Ida - o conjunto de objetos da primeira derivação - é armazenado na memória RAM. Finalmente, o conjunto de objetos é processado no bloco BC_Volta, como mostra a Figura 12.

Na Figura 14 é apresentado um diagrama que mostra o processo do algoritmo em um *Work Item*. Neste diagrama é exibido a entrada $A[i]$ enviada por parâmetro pelo

Figura 13 – Arquitetura OpenCL



Fonte: Autoria própria

kernel. Em seguida, o bloco BC_Ida realiza a derivação de atributos, produzindo o conjunto de objetos. Na sequência, o resultado dessa derivação é armazenado em outra execução do processo de derivação, agora usando o conjunto de objetos como entrada. O resultado dessa derivação é armazenado em uma matriz Ar. Finalmente, o resultado dessa derivação $(A[i]r)$ é comparado com a entrada $(A[i])$ para confirmar se é ou não um conceito formal. Essa comparação é realizada usando uma estratégia *bit* a *bit*. Após o processamento de todos os *Work Item*, o resultado é armazenado em uma variável de retorno, que será enviada de volta ao *host*.

Figura 14 – Algoritmo de Força Bruta

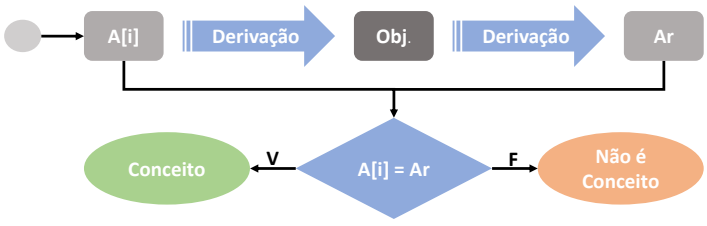


Tabela de Contexto

Obj.	Atributos			
	A	B	C	D
O1	X	X	X	
O2		X	X	
O3	X		X	
O4		X		X

Legenda

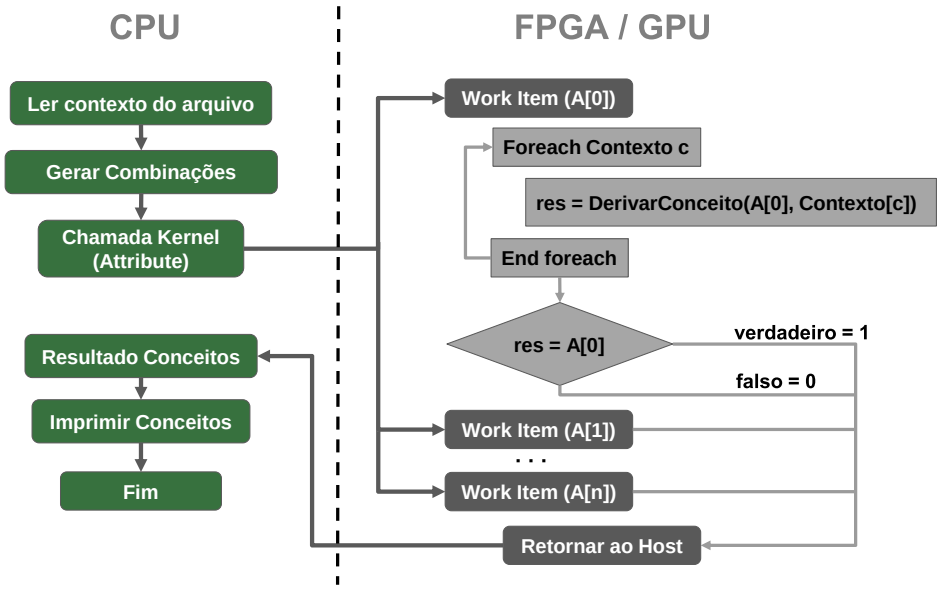
A[i]	Atributos de entrada
Obj.	Resultado da derivação dos atributos
Ar	Resultado da derivação de objetos
	Função de derivação

Fonte: Autoria própria

Vale ressaltar que, para arquiteturas heterogêneas, o algoritmo de Força Bruta é executado em paralelo para o FPGA e a GPU. Na Figura 15 é possível ver que o

host chama o *kernel* passando os atributos e o contexto como parâmetros. O dispositivo distribui a execução de cada combinação de atributos para todos os *Work Item* disponíveis. Após o processamento de cada *Work Item*, os conceitos são armazenados para serem enviados de volta ao *host*.

Figura 15 – Algoritmo de Força Bruta Paralelo - Arquitetura Heterogênea



Fonte: Autoria própria

6 RESULTADOS EXPERIMENTAIS

Assim como apresentado na Seção 4.3, os algoritmos de Força Bruta (desenvolvidos neste trabalho) e o algoritmo *Data-Peeler* foram avaliados de forma que os resultados são discutidos em três categorias: desempenho (Seção 6.1), consumo de energia (Seção 6.2) e Operações por *Watt* (Seção 6.3).

6.1 Desempenho

Nesta seção é apresentado o tempo de execução, em segundos (s), para cada contexto processado. A Tabela 2 mostra esses resultados obtidos para o algoritmo de Força Bruta em todas as densidades. Para fins de comparação, a Tabela 7 apresenta os respectivos resultados para o algoritmo *Data-Peeler*.

Após realizar a primeira execução de testes, é possível observar que o tempo de execução dos algoritmos de Força Bruta não dependem da densidade. Isso ocorre porque todas as combinações possíveis de atributos são processadas nessa estratégia. Assim, o tempo de processamento desse algoritmo está vinculado apenas ao número de objetos e atributos. Por outro lado, o tempo de processamento do *Data-Peeler* também depende da densidade.

Tabela 2 – Tempo de execução (segundos) - Força Bruta

Objetos	**CPU**		**CPU+FPGA**	**CPU+GPU**
	24 Threads	**88 Threads**		
1.000	11,45	2,21	125,16	0,52
5.000	55,49	10,02	603,74	1,81
10.000	107,71	17,32	1.027,25	3,44
50.000	535,21	85,22	4.871,36	16,38
100.000	1.077,62	184,15	9.983,41	32,82
500.000	5.385,47	920,52	49.765,29	169,38
1.000.000	10.792,12	1.912,07	99.735,08	338,61

Na Tabela 2, nota-se que o algoritmo de Força Bruta funciona bem para processadores com o maior número de núcleos. Isso se deve à sua baixa complexidade e às suas altas características de dissociação, requisitos fundamentais para escalabilidade. Assim, quanto maior o número de núcleos de processamento, maior o desempenho dessa abordagem. Além disso, esse comportamento ocorre em abordagens homogêneas e heterogêneas.

O tempo de execução da abordagem CPU + GPU foi o mais baixo entre os ambientes. Isso aconteceu porque o número de unidades de processamento nas GPUs são maiores que no FPGA ou na própria CPU. Esta declaração está relacionada à quantidade de *Work Item* instanciados, que está diretamente vinculada ao número de unidades de processamento.

Considerando que a abordagem CPU + GPU apresentou melhores resultados que as demais, este é destacado para comparações com o *Data-Peeler*. Portanto, foi variado o número de atributos e densidade para comparação com o algoritmo *Data-Peeler*. A Tabela 3 apresenta o tempo de execução para o número de atributos e objetos da arquitetura heterogênea da CPU + GPU. Da mesma forma, é apresentado nas Tabelas 4, 5 e 6 os tempos de execução do *Data-Peeler*, variando os atributos para as três densidades (30%, 50% e 70%).

Tabela 3 – Tempo de execução (s) - CPU+GPU

Objetos	**Atributos**					
	25	**26**	**27**	**28**	**29**	**30**
1.000	0,52	1,21	2,49	4,22	10,42	17,86
5.000	1,81	5,41	10,91	17,39	43,89	62,52
10.000	3,44	10,67	22,01	36,98	91,28	118,04
50.000	16,38	50,86	109,05	185,09	425,59	590,21
100.000	32,82	100,11	214,15	373,81	927,78	1.191,04
500.000	169,38	535,45	1.098,53	1.739,04	4.609,01	6.177,47
1.000.000	338,61	1.035,19	2.101,21	3.698,61	9.278,13	11.983,63

Tabela 4 – Tempo de execução (s) - Data-Peeler 30% de Densidade

Objetos	**Atributos**					
	25	**26**	**27**	**28**	**29**	**30**
1.000	0,69	0,72	0,88	0,91	1,15	1,95
5.000	3,82	3,64	4,69	10,3	14,51	32,95
10.000	7,62	8,74	9,62	17,3	23,72	45,65
50.000	39,74	41,79	59,56	73,87	98,16	121,71
100.000	99,03	102,01	187,97	231,99	311,87	384,09
500.000	1.466,11	1.649,33	1.798,21	1.912,95	2.108,37	2.742,67
1.000.000	6.399,13	7.013,19	7.460,51	8.221,13	9.624,87	10.708,15

As Tabelas 5 e 6 apresentam os resultados para a variação de atributos do *Data-Peeler*, com densidades de 50% e 70%, respectivamente. Sendo assim, é possível observar uma inconsistência ao obter resultados com um número maior de objetos com esse algoritmo. Seu tempo de execução se torna impraticável, pois o excesso de memória ocorreu durante a execução. Portanto, os resultados dessas execuções não são mostrados, sendo representados pelo símbolo *.

Tabela 5 – Tempo de execução (s) - Data-Peeler 50% de Densidade

Objetos	Atributos					
	25	**26**	**27**	**28**	**29**	**30**
1.000	16,65	23,48	31,93	43,16	59,77	79,56
5.000	91,69	154,21	219,49	338,31	486,23	699,88
10.000	182,92	273,61	428,02	675,89	*	*
50.000	953,78	1.641,66	2.146,76	3.717,92	*	*
100.000	2.376,82	*	*	*	*	*
500.000	*	*	*	*	*	*
1.000.000	*	*	*	*	*	*

Tabela 6 – Tempo de execução (s) - Data-Peeler 70% de Densidade

Objetos	Atributos					
	25	**26**	**27**	**28**	**29**	**30**
1.000	399,71	732,53	866,32	1.079,35	2.355,62	4.384,12
5.000	2.200,50	3.701,04	5.701,04	*	*	*
10.000	4.390,14	6.566,64	*	*	*	*
50.000	*	*	*	*	*	*
100.000	*	*	*	*	*	*
500.000	*	*	*	*	*	*
1.000.000	*	*	*	*	*	*

Logo, é possível concluir que, o *Data-Peeler* apresenta melhores resultados que o Força Bruta para baixas densidades e maior número de atributos. No entanto, vale ressaltar que o Força Bruta é mais escalável que o *Data-Peeler*. Assim, podemos aumentar o desempenho aumentando o número de núcleos de processamento com o Força Bruta. Contudo, o *Data-Peeler* não altera o tempo de processamento, independentemente do aumento de núcleos, pois é um código sequencial. Para densidades mais altas, o Força Bruta tem um desempenho superior ao *Data-Peeler*. Isso independentemente do número de atributos, pois ele pode processar todas as combinações sem ser sobrecarregado pelo nível de densidade.

Continuando a avaliação, foi realizado novas execuções mantendo o número de atributos igual a 25. Assim, é possível detalhar melhor a comparação entre os algoritmos de Força Bruta e *Data-Peeler*. Os testes foram realizados com as mesmas densidades e dados de entrada para ambos os algoritmos. Novamente, o tempo de execução do *Data-Peeler* é influenciado não apenas pelo número de objetos e atributos, mas também pela densidade. A Tabela 7 mostra o tempo de execução do *Data-Peeler* neste último cenário.

O tempo de execução se torna impraticável com o *Data-Peeler* acima de dois limites: mais de 100.000 objetos para 50% de densidade e mais de 50.000 objetos para 70% de densidade. Além disso, os recursos de memória eram insuficientes nos ambientes

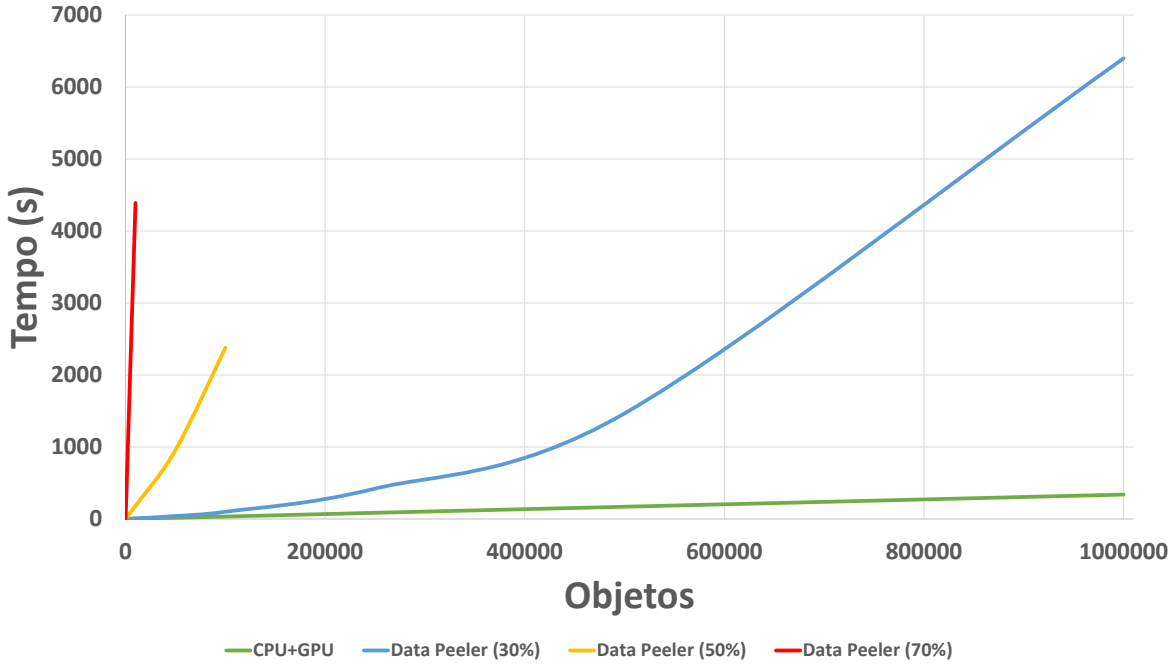
usados com esses conjuntos de dados de entrada. Assim, foi possível concluir que o *Data-Peeler* não apresenta um bom desempenho para conjuntos de dados com altas densidades e tamanhos.

Tabela 7 – Tempo de execução (s) - Data-Peeler

Objetos	Data-Peeler (CPU)		
	30%	**50%**	**70%**
1.000	0,69	16,65	399,71
5.000	3,82	91,69	2.200,50
10.000	7,62	182,92	4.390,14
50.000	39,74	953,78	*
100.000	99,03	2.376,82	*
500.000	1.466,11	*	*
1.000.000	6.399,13	*	*

Na Figura 16 foi plotado o gráfico para avaliar a proposta deste trabalho. Separando os resultados do algoritmo de Força Bruta quando executados no ambiente CPU + GPU. Esse foi o que apresentou melhores resultados após análises anteriores.

Figura 16 – Tempo de execução (segundos)



Fonte: Autoria própria

Após as verificações, foi possível concluir que, o algoritmo de Força Bruta possui um desempenho melhor nessas conduções de execução que o *Data-Peeler*, mesmo quando há densidades variadas. Os resultados do Força Bruta foram 18×, 74× e 1.000× melhores

que o *Data-Peeler* com, respectivamente, 30%, 50% e 70% de densidade. Além disso, o algoritmo de Força Bruta mantém um comportamento constante no tempo de processamento. Para 10.000 objetos, são gastos aproximadamente 3 segundos. Para 100.000 objetos, o tempo foi aumentado em aproximadamente 10× (32 segundos), além de 1.000.000 objetos (338 segundos). O aumento no número de objetos está diretamente relacionado ao aumento no tempo de execução. Além disso, não alteramos o número de atributos e, ao contrário do *Data-Peeler*, a densidade pode ser negligenciada na Força Bruta.

6.2 Consumo de Energia

Nesta seção, é mostrado o consumo de energia dos testes, medido em *Joules* (J). Os valores de potência, em *Watts*, foram obtidos das ferramentas *Powertop* (CPU), *ModelSIM* (FPGA) e *CUDA Toolkit* 10 (GPU) para a CPU, FPGA e GPU, respectivamente. O consumo de energia foi, então, calculado usando a fórmula $E = P \times t$, em que E corresponde à energia, P à potência do dispositivo e t ao tempo de execução. Considerando que o aumento do tempo de execução seja inviável para densidades acima de 50% (quando se trata do *Data-Peeler*) principalmente para atributos maiores, nesta seção foi selecionado apenas os resultados com 25 atributos para análise. Na estratégia homogênea, usando apenas a CPU, foi utilizado os valores de potência obtidos pelo *Powertop* em conjunto ao tempo de execução. Na abordagem heterogênea, foi adicionado o consumo de energia da CPU ao dispositivo, obtendo assim o consumo geral de energia do sistema.

As Tabelas 8 e 9 mostram o consumo de energia dos algoritmos de Força Bruta e *Data-Peeler*, respectivamente. O FPGA é um dispositivo que consome pouca energia, devido à sua baixa frequência e memória restrita. No entanto, quando o consumo do FPGA é adicionado à CPU, o consumo se torna maior que o esperado. Para a GPU, apesar de seu consumo de energia ser maior do que no FPGA, o consumo total de energia - CPU adicionada - diminui devido ao tempo de execução, que é muito menor. Lembrando que as GPUs têm maior frequência e número de núcleos de processamento que o FPGA, consumindo mais energia e oferecendo computação mais rápida.

Especificamente para o *Data-Peeler*, o consumo de energia na abordagem somente de CPU não foi mostrado completamente para densidades acima de 50%. Não foi possível executar esse algoritmo para um grande número de objetos, portanto, seu tempo de execução não pôde ser obtido.

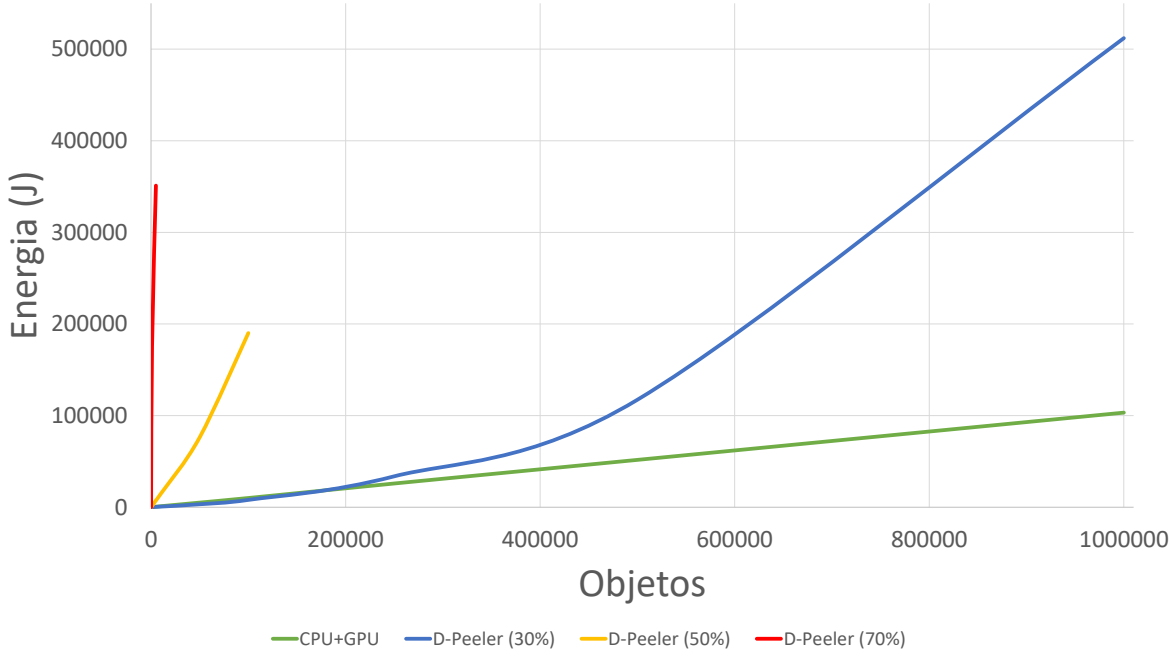
Tabela 8 – Consumo de energia (J) - Força Bruta

Objetos	CPU		CPU+FPGA	CPU+GPU
	24 Threads	88 Threads		
1.000	916,0	176,8	10.513,4	158,6
5.000	4.439,2	801,6	50.714,2	552,1
10.000	8.616,8	1.385,6	86.289,0	1.049,2
50.000	42.816,8	6.817,6	409.194,2	4.995,9
100.000	86.209,6	14.732,0	838.606,4	10.010,1
500.000	430.837,6	73.641,6	4.180.284,4	51.660,9
1.000.000	863.369,6	152.965,6	8.377.746,7	103.276,1

Tabela 9 – Consumo de energia (J) - Data-Peeler

Objetos	Data-Peeler		
	30%	50%	70%
1.000	55,2	1.324,8	176.025,6
5.000	305,6	7.334,4	351.129,6
10.000	609,6	14.630,4	*
50.000	3.179,2	76.300,8	*
100.000	7.922,4	190.137,6	*
500.000	117.288,8	*	*
1.000.000	511.930,4	*	*

Figura 17 – Consumo de energia



Fonte: Autoria própria

Para o Força Bruta, o consumo de energia na abordagem CPU + GPU apresentou os melhores resultados. Portanto, este foi usado para comparação com os resultados das execuções do *Data-Peeler*. A figura 17 mostra essa comparação. Contudo, foi observado que o consumo de energia do *Data-Peeler* na densidade de 30% compete com o Força Bruta ao usar poucos objetos. No entanto, à medida que o número de objetos aumenta, o consumo de energia se torna maior na plataforma heterogênea.

A arquitetura heterogênea usa dois componentes (*host* e *device*). Assim, o consumo de energia de cada dispositivo é adicionado e depois multiplicado pelo tempo de execução. Por outro lado, para o *Data-Peeler* com 30% de densidade, o consumo de energia refere-se apenas à CPU. Como o tempo de execução foi próximo ao algoritmo de Força Bruta (CPU + GPU) para poucos objetos, e a CPU consome menos energia que a CPU + GPU, o consumo geral de energia foi menor. Após avaliação, o consumo de energia está mais ligado ao tempo de execução. Assim, para densidades acima de 50%, o *Data-Peeler* se torna o maior consumidor de energia do que a abordagem do Força Bruta.

6.3 Eficiência Energética

Para avaliar a eficiência energética de cada abordagem, foi analisado a métrica de Milhar de bilhão de Operações por *Watt*, ou seja, PetaOps/ *Watt*. Essa métrica consiste em uma razão entre todas as operações aritméticas e lógicas que o algoritmo faz e divide pela multiplicação do tempo com a potência (energia). Assim, quanto maior as Operações/ *Watt*, melhor a eficiência energética do sistema.

Considerando que as métricas para calculo de operações/ *Watt* estão ligadas diretamente ao algoritmo desenvolvido, em conjunto ao tempo de processamento, os resultados obtidos para eficiência energética, não são influenciados pela variação dos objetos ou atributos. Logo, para obter e comparar estes resultados, foram avaliados as variações de objetos e fixado a quantidade de atributos em 25.

Sendo assim, é mostrado a média aritmética após 10 execuções. Para os algoritmos de Força Bruta e *Data-Peeler*, respectivamente, mais de 90% e 80% dos testes estão dentro do desvio padrão.

Tabela 10 – Eficiência energética

Dispositivos	**PetaOps/Watt**
CPU	3,42
CPU+FPGA	0,08
CPU+GPU	6,70
Data-Peeler (CPU)	3,74

A Tabela 10 mostra os PetaOps/*Watt* obtidos para cada arquitetura. A estratégia que atinge a maior eficiência energética é a CPU + GPU. Em outras palavras, ele executa mais operações por segundo por *Watt* do que outras estratégias avaliadas. A abordagem CPU + GPU executa a maioria de suas instruções em paralelo. Esse é o principal fator por ser 2× melhor que o *Data-Peeler* e 83× melhor que a abordagem CPU + FPGA.

Além disso, vale ressaltar que esse resultado está diretamente relacionado à quantidade de instruções para cada algoritmo juntamente com seu gasto de energia, ou seja, quanto maior a complexidade do algoritmo e energia, maior as operações/*Watt*. Assim, abordagens contendo poucas instruções e baixo consumo tendem a realizar mais operações/*Watt* e apresentar melhores desempenhos.

7 CONCLUSÕES

A área de mineração de dados requer computação de alto desempenho. As pesquisas geralmente carecem de recursos computacionais capazes de processar algoritmos atuais em seu estado atual. No entanto, há um fator decisivo no uso ou não de uma tecnologia, ferramenta ou algoritmo, uma vez que máquinas contendo um processador de alto desempenho podem ser muito caras. Neste trabalho, foi usado abordagens homogêneas e heterogêneas para executar implementações de algoritmos de Análise Formal de Conceitos (FCA). Foi possível mostrar que a computação heterogênea pode ser usada para obter um melhor desempenho que não é alcançado com arquiteturas homogêneas.

Um dos objetivos da área de FCA é processar um contexto formal e extrair conceitos formais. O desempenho é medido pelo tempo necessário para esse processo. Assim, foi apresentado neste trabalho o desempenho de uma arquitetura heterogênea de CPU + GPU ao executar uma abordagem com um algoritmo de Força Bruta. Ele provou ser mais rápido que as arquiteturas homogêneas, mesmo quando comparado com algoritmos mais sofisticados, atingindo um *Speedup* de 18×.

Além do desempenho, foi avaliado o consumo de energia gasto pelo sistema e a eficiência das operações por segundo por *Watts*. A arquitetura CPU + GPU apresentou pelo menos 2× maior eficiência do que outras arquiteturas. Além disso, essa estratégia apresentou menor consumo de energia do que outras abordagens para o algoritmo de Força Bruta. Por fim, esta é mais eficiente que o *Data-Peeler*, um algoritmo da área de FCA de última geração, para grandes quantidades de objetos com densidades acima de 30%.

Devido ao alto desacoplamento do algoritmo de Força Bruta, ele é bem dimensionado com o aumento do número de unidades de processamento. Portanto, quanto maior a quantidade de *Work Item* em uma abordagem que utiliza *OpenCL*, menor o tempo para processar um grande número de combinações de atributos. Dito isto, é possível concluir que a estratégia do Força Bruta desenvolvida neste trabalho é escalável, principalmente em relação ao número de *Work Item*.

Para trabalhos futuros, planeja-se explorar arquiteturas paralelas para desenvolvimento de códigos em *OpenCL* com algoritmos mais sofisticados, por exemplo, o *Data-Peeler*. Também propõe-se a implementação de algoritmos para paralelizar tanto o conjunto de objetos quanto o de atributos, visando uma melhor distribuição das possíveis combinações. Contudo, é esperado superar a complexidade desses algoritmos, a fim de evitar comparações desnecessárias e obter melhor desempenho.

REFERÊNCIAS

ANDRADE, H.; CRNKOVIC, I. A review on software architectures for heterogeneous platforms. In: IEEE. 2018 25th Asia-Pacific Software Engineering Conference (APSEC). [S.l.], 2018. p. 209–218.

ANDREWS, S. In-close2, a high performance formal concept miner. In: SPRINGER. International Conference on Conceptual Structures. [S.l.], 2011. p. 50–62.

ANDREWS, S. A 'best-of-breed'approach for designing a fast algorithm for computing fixpoints of galois connections. Information Sciences, Elsevier, v. 295, p. 633–649, 2015.

ANDREWS, S.; ORPHANIDES, C. Analysis of large data sets using formal concept lattices. University of Seville, 2010.

BRODTKORB, A. R. et al. State-of-the-art in heterogeneous computing. Scientific Programming, Hindawi, v. 18, n. 1, p. 1–33, 2010.

CASTRO, G. T. et al. A fast parallel k-modes algorithm for clustering nucleotide sequences to predict translation initiation sites. Journal of Computational Biology, Mary Ann Liebert, Inc., publishers 140 Huguenot Street, 3rd Floor New ..., v. 26, n. 5, p. 442–456, 2019.

CERF, L. et al. Closed and noise-tolerant patterns in n-ary relations. Data Mining and Knowledge Discovery, Springer, v. 26, n. 3, p. 574–619, 2013.

CERF, L. et al. Data-peeler: Constraint-based closed pattern mining in n-ary relations. In: SIAM. proceedings of the 2008 SIAM International conference on Data Mining. [S.l.], 2008. p. 37–48.

CERF, L. et al. Closed patterns meet n-ary relations. ACM Transactions on Knowledge Discovery from Data (TKDD), ACM New York, NY, USA, v. 3, n. 1, p. 1–36, 2009.

CHEN, T. et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM. ACM Sigplan Notices. [S.l.], 2014. v. 49, n. 4, p. 269–284.

DAVEY, B.; PRIESTLEY, H. Introduction to lattices and order cambridge univ. Press, Cambridge, 1990.

DONGARRA, J.; LASTOVETSKY, A. L. High performance heterogeneous computing. [S.l.]: John Wiley & Sons, 2009.

FAGGIN, F. et al. The history of the 4004. Ieee Micro, IEEE, v. 16, n. 6, p. 10–20, 1996.

FLYNN, M. J. Very high-speed computing systems. Proceedings of the IEEE, IEEE, v. 54, n. 12, p. 1901–1909, 1966.

GANTER, B.; RUDOLPH, S.; STUMME, G. Explaining data with formal concept analysis. In: Reasoning Web. Explainable Artificial Intelligence. [S.l.]: Springer, 2019. p. 153–195.

GANTER, B.; WILLE, R. Formal concept analysis: mathematical foundations. [S.l.]: Springer Science & Business Media, 2012.

GRÄTZER, G. General lattice theory. [S.l.]: Springer Science & Business Media, 2002.

GUTIÉRREZ, P. D. et al. Smote-gpu: Big data preprocessing on commodity hardware for imbalanced classification. Progress in Artificial Intelligence, Springer, v. 6, n. 4, p. 347–354, 2017.

HAO, F. et al. k-cliques mining in dynamic social networks based on triadic formal concept analysis. Neurocomputing, Elsevier, v. 209, p. 57–66, 2016.

HENRIQUES, R.; MADEIRA, S. C. Triclustering algorithms for three-dimensional data analysis: a comprehensive survey. ACM Computing Surveys (CSUR), ACM New York, NY, USA, v. 51, n. 5, p. 1–43, 2018.

JALALI, S. et al. A comparative analysis of classifiers in cancer prediction using multiple data mining techniques. A comparative analysis of classifiers in cancer prediction using multiple data mining techniques, Inderscience, n. 2, p. 166–178, 2017.

KANG, C. Y. et al. Hardware processors and methods for tightly-coupled heterogeneous computing. [S.l.]: Google Patents, jan. 16 2018. US Patent 9,870,339.

KODAGODA, N.; ANDREWS, S.; PULASINGHE, K. A parallel version of the in-close algorithm. In: IEEE. 2017 6th National Conference on Technology and Management (NCTM). [S.l.], 2017. p. 1–5.

KUMAR, S. et al. A network on chip architecture and design methodology. In: IEEE. Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002. [S.l.], 2002. p. 117–124.

MACIEL, L. et al. Arquitetura heterogênea cpu+ fpga para análise formal de conceitos. In: SBC. Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho. [S.l.], 2019. p. 85–96.

MACIEL, L. A.; SOUZA, M. A.; FREITAS, H. C. Reconfigurable fpga-based k-means/k-modes architecture for network intrusion detection. IEEE Transactions on Circuits and Systems II: Express Briefs, IEEE, p. 1–5, 2019.

MISSAOUI, R.; KUZNETSOV, S. O.; OBIEDKOV, S. Formal Concept Analysis of Social Networks. [S.l.]: Springer, 2017.

MORAES, N. R. de et al. Parallelization of the next closure algorithm for generating the minimum set of implication rules. Artif. Intell. Research, v. 5, n. 2, p. 40–54, 2016.

NEELIMA, C.; SARMA, S. S. Blended intelligence of fca with flc for knowledge representation from clustered data in medical analysis. International Journal of Electrical and Computer Engineering, IAES Institute of Advanced Engineering and Science, v. 9, n. 1, p. 635, 2019.

NERI, M. et al. Lei de moore e políticas de inclusão digital. Revista Inteligência Empresarial, Rio de Janeiro, v. 24, n. 14, p. 5, 2003.

NESHATPOUR, K. et al. Accelerating big data analytics using fpgas. In: IEEE. 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. [S.l.], 2015. p. 164–164.

NETO, S. M. et al. Identification of substructures in complex networks using formal concept analysis. International Journal of Web Information Systems, Emerald Publishing Limited, v. 14, n. 3, p. 281–298, 2018.

NETO, S. M.; ZÁRATE, L. E.; SONG, M. A. Handling high dimensionality contexts in formal concept analysis via binary decision diagrams. Information Sciences, Elsevier, v. 429, p. 361–376, 2018.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Computer Architecture News, ACM, v. 42, n. 3, p. 13–24, 2014.

RODRÍGUEZ-LORENZO, E. et al. An axiomatic system for conditional attribute implications in triadic concept analysis. International Journal of Intelligent Systems, Wiley Online Library, v. 32, n. 8, p. 760–777, 2017.

ROSE, C. D.; NAVAUX, P. Fundamentos de processamento de alto desempenho. Anais: 2a Escola Regional de Alto Desempenho, p. 3–29, 2002.

SANTOS, P. et al. Implicpbdd: A new approach to extract proper implications set from high-dimension formal contexts using a binary decision diagram. Information, Multidisciplinary Digital Publishing Institute, v. 9, n. 11, p. 266, 2018.

SILVEIRA, C. L.; JR, L. G. da S.; CAVALHEIRO, G. G. H. Programação em opencl: Uma introdução prática. 2010.

SOUZA, M. A. et al. Energy efficient parallel k-means clustering for an intel® hybrid multi-chip package. In: . [S.l.]: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2018. p. 372–379. ISSN 1550-6533.

STRINGHINI, D.; GONÇALVES, R. A.; GOLDMAN, A. Introdução à computação heterogênea. In: Anais da XXXI Jornada de Atualização em Informática do XXXII Congresso da Sociedade Brasileira de Computação [Internet]. [S.l.: s.n.], 2012. p. 16–19.

WILLE, R. Restructuring lattice theory: an approach based on hierarchies of concepts. In: SPRINGER. International Conference on Formal Concept Analysis. [S.l.], 2009. p. 314–339.

APÊNDICE A – ALGORITMO GPU

Código A.1 – Código do Host da GPU

```
1  #include <iostream>
2  #include <cstdio>
3  #include <cstring>
4  #include <time.h>
5  #include <CL/cl.hpp>
6  #include <string>
7  #include <fstream>
8  #include <math.h>
9  #include <errno.h>
10 #include <stdio.h>
11 #include <ctime>
12
13 #define CL_USE_DEPRECATED_OPENCL_2_0_APIS
14
15 int *dadosROM;
16 int *retKernel;
17
18 int qtdObjetos = 1000000;
19 int combAtributos = 33554431;
20
21 void CarregaContexto(){
22   dadosROM = (int*)malloc(sizeof(int) * qtdObjetos);
23   retKernel = (int*)malloc(sizeof(int) * combAtributos);
24
25   for (int w = 0; w < combAtributos; w++)
26   {
27     retKernel[w] = 0;
28   }
29   char dir[200];
30
31   char Linha[25];
32   char *result;
33
34   FILE* arq = fopen("baseTeste.txt", "r");
35
36   if (arq == NULL)
37   {
38       printf("Problemas na CRIACAO do arquivo\n");
39       return;
```

```
40    }
41
42    int i = 0;
43    while (!feof(arq))
44    {
45      result = fgets(Linha, 25, arq);
46      if (result){
47        dadosROM[i] = atoi(Linha);
48      }
49      i++;
50    }
51    fclose(arq);
52  }
53
54  void SalvarResultadoArquivo(double tempoGasto, int conceitos){
55    char palavra[200] = "Tempo gasto: ";
56
57    FILE* pont_arq = fopen("baseTeste.txt", "r");
58
59    if(pont_arq != NULL){
60      fprintf(pont_arq, "%s", "Testes de ");
61      fprintf(pont_arq, "%d", qtdObjetos);
62      fprintf(pont_arq, "%s", " Objetos e ");
63      fprintf(pont_arq, "%d", combAtributos);
64      fprintf(pont_arq, "%s", " Combinacoes\n");
65      fprintf(pont_arq, "%s", "Tempo gasto: ");
66      fprintf(pont_arq, "%lf", tempoGasto);
67      fprintf(pont_arq, "%s", "  - Conceitos: ");
68      fprintf(pont_arq, "%d \n\n", conceitos);
69    }
70
71
72    fclose(pont_arq);
73  }
74
75  int main()
76  {
77     printf("\nDigite a quantidade de objetos e atributos");
78     scanf("%d",qtdObjetos);
79     scanf("%d",combAtributos);
80
81     printf("\nIniciando programa...");
82     //Declarao de varivel responsvel por medir o tempo
83     clock_t time_clock;
84     int Conceitos = 0;
85     cl_int err = 0;
86     double Tempo;
```

```
87
88
89      //Criando o programa
90      std::vector<cl::Platform> platforms;
91      cl::Platform::get(&platforms);
92
93      cl::Platform platform = platforms[1];
94      std::vector<cl::Device> devices;
95      platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);
96
97      cl::Device device = devices.front();
98
99      std::ifstream helloWorldFile("hello_world.cl");
100     std::string src(std::istreambuf_iterator<char>(helloWorldFile), (std::
            istreambuf_iterator<char>()));
101
102     cl::Program::Sources sources(1, std::make_pair(src.c_str(), src.length()
            + 1));
103
104     cl::Context context(device);
105
106     cl::Program program(context, sources);
107
108     program.build("-cl-std=CL1.2");
109
110     //Carregando os dados do arquivo
111     printf("\nCarregando contexto...");
112     CarregaContexto();
113
114     //inicializando os buffers e parametros
115     printf("\nIniciando Kernel...");
116     cl::Buffer dadosBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            sizeof(int) * qtdObjetos, dadosROM, &err);
117     cl::Buffer retornoBuffer(context, CL_MEM_READ_WRITE |
           CL_MEM_COPY_HOST_PTR, sizeof(int) * combAtributos, retKernel, &err);
118     cl::Kernel kernel(program, "hello_world");
119
120     err = kernel.setArg(0, dadosBuffer);
121     err = kernel.setArg(1, retornoBuffer);
122     err = kernel.setArg(2, qtdObjetos);
123     err = kernel.setArg(3, combAtributos);
124
125     cl::CommandQueue queue(context, device);
126
127     printf("\nExecutando...\n");
128
129     //executando o kernel
```

```
130     time_clock = clock();
131     err = queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(
            combAtributos));
132
133
134     //lendo o resultado do kernel
135     err = queue.enqueueReadBuffer(retornoBuffer, CL_TRUE, 0, sizeof(int)*
            combAtributos, retKernel);
136
137     //contar a quantidade de conceitos
138     for(int z=0; z<combAtributos; z++)
139     {
140        Conceitos = Conceitos + retKernel[z];
141     }
142     Conceitos += 2;
143     time_clock = clock() - time_clock;
144
145     printf("\n\nTotal de Conceitos: %d\n\n", Conceitos);
146
147     printf("Tempo gasto: %f\n\n", ((float)time_clock) / CLOCKS_PER_SEC );
148     //gravar em um arquivo o resultado
149     SalvarResultadoArquivo(((float)time_clock) / CLOCKS_PER_SEC, Conceitos);
150
151     std::cin.get();
152
153 }
```

Código A.2 – Código do Device para GPU e FPGA

```
1  __kernel void analisarConceito(__global int * restrict dadosROM,
2                                 __global int * restrict retKernel,
3                                 const int qtdObjetos,
4                                 const int combAtributos) {
5
6      //çõDeclaraes
7      int RX = combAtributos;
8      int i,j = 0;
9      int filtro_id = get_global_id(0);
10
11     if (filtro_id > 0 && filtro_id < RX)
12     {
13         for(i=0; i<qtdObjetos; i++) //Percorre todos os objetos da base (
               ómemria ROM)
14         {
15             //verifica se os resultados do "AND" é igual ao filtro para
                   entrar na RAM de objetos do óprximo passo
16             if((filtro_id & dadosROM[i]) == filtro_id)
```

```
17              {
18                  RX = RX & dadosROM[i];
19              }
20              else
21              {
22                  j++;
23              }
24          }
25
26          //Caso o resultado do AND bit a bit dfor igual ao Filtro de entrada
                , é um conceito, armazenando no vetor correspondente
27          if((RX == filtro_id) && (j != qtdObjetos))
28          {
29             retKernel[filtro_id]=1;
30          }
31      }
32  }
```

APÊNDICE B – ALGORITMO FPGA

Código B.1 – Código do Host do FPGA

```
1  /*
2   * Meant to be used as template for new applications...
3   * Recommended to refer to OpenCL at Khronos:
4   *     https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/
5   * Recommended to refer to Intel Altera OpenCL SDK for FPGA Programming
        Guide
6   * Recommended to refer to Intel Altera OpenCL SDK for FPGA Best Practices
        Guide
7   *
8   * Template HOST to call simple kernel.
9   */
10
11 #include <assert.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <math.h>
15 #include <cstring>
16 #include "CL/opencl.h"
17 #include "AOCLUtils/aocl_utils.h"
18 #include <unistd.h>
19 #include <string.h>
20
21 using namespace aocl_utils;
22
23 // OpenCL runtime configuration
24 static cl_platform_id platform = NULL;
25 static cl_device_id device = NULL;
26 static cl_context context = NULL;
27 static cl_command_queue queue = NULL;
28 static cl_kernel kernel = NULL;
29 static cl_program program = NULL;
30
31 cl_int status;
32
33 // Function prototypes
34 bool init();
35 void cleanup();
36
37 //-------------------------------------------------------------------------
```

```
38 int *dadosROM;
39 int *retKernel;
40 int *RamObjetos;
41 int filtro;
42
43 int qtdObjetos = 1000;
44 int combAtributos = 33554431;
45
46
47 cl_mem dadosBuffer, retornoBuffer, RamObjetosBuffer;
48
49 //------------------------------------------------------------------------
50
51
52 void CreateKernel(void) {
53     kernel = clCreateKernel(program, "analisarConceito" , &status);
54     checkError(status, "Failed to create kernel");
55 }
56
57 void FinalizarKernel(){
58   status = clFinish(queue);  // Wait the queue
59   checkError(status, "Failed to finish");
60   printf("\nKernel execution is complete.\n");
61 }
62
63 void CarregaContexto(){
64   dadosROM = (int*)malloc(sizeof(int) * qtdObjetos);
65   retKernel = (int*)malloc(sizeof(int) * combAtributos);
66
67   for (int w = 0; w < combAtributos; w++)
68   {
69     retKernel[w] = 0;
70   }
71   char dir[200];
72
73   strcpy(dir, get_current_dir_name());
74
75   int aux = strlen(dir);
76   char Linha[25];
77   char *result;
78
79   strcat(dir, "/baseTeste.txt");
80
81   FILE *arq = fopen(dir, "rt");
82
83   if (arq == NULL)
84   {
```

```
85        printf("Problemas na CRIACAO do arquivo\n");
86        return;
87    }
88
89    int i = 0;
90    while (!feof(arq))
91    {
92      result = fgets(Linha, 25, arq);
93      if (result){
94        dadosROM[i] = atoi(Linha);
95      }
96      i++;
97    }
98    fclose(arq);
99  }
100
101 void StartKernel(){
102   //criando os buffers
103   dadosBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR,
104                 sizeof(int) * qtdObjetos, dadosROM, &status);
105   retornoBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE |
        CL_MEM_COPY_HOST_PTR,
106                   sizeof(int) * combAtributos, retKernel, &status);
107
108   //escrevendo o contexto no buffer
109   status  = clEnqueueWriteBuffer(queue, dadosBuffer, CL_FALSE, 0,
110             sizeof(int) * qtdObjetos, dadosROM, 0, NULL, NULL);
111   checkError(status, "Falhou ao enfileirar buffer");
112
113   //set nas ávariveis do kernel
114   status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dadosBuffer);
115   status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &retornoBuffer);
116   status |= clSetKernelArg(kernel, 2, sizeof(int), &qtdObjetos);
117   status |= clSetKernelArg(kernel, 3, sizeof(int), &combAtributos);
118
119   checkError(status, "Falhou ao setar argumentos");
120
121   status |= clFinish(queue);
122    checkError(status, "Falhou ao finalizar");
123 }
124
125 void ExecutarKernel(){
126   size_t size[3] = { combAtributos , 1 , 1 };
127   status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, size, &size[0],
        0, NULL, NULL);
128   checkError(status, "Falha ao executar o kernel");
```

```
129
130   status |= clEnqueueReadBuffer(queue, retornoBuffer, CL_TRUE, 0, sizeof(
          int) * combAtributos, retKernel, 0, NULL, NULL);
131   checkError(status, "Falha ao ler os Buffers");
132
133 }
134
135 void SalvarResultadoArquivo(double tempoGasto, int conceitos){
136   FILE *pont_arq;
137   char palavra[200] = "Tempo gasto: "; // ávarivel do tipo string
138
139   //abrindo o arquivo
140   pont_arq = fopen("arquivoResultado.txt", "a");
141
142   if(pont_arq != NULL){
143     fprintf(pont_arq, "%s", "Testes de ");
144     fprintf(pont_arq, "%d", qtdObjetos);
145     fprintf(pont_arq, "%s", " Objetos e ");
146     fprintf(pont_arq, "%d", combAtributos);
147     fprintf(pont_arq, "%s", " Combinacoes\n");
148     fprintf(pont_arq, "%s", "Tempo gasto: ");
149     fprintf(pont_arq, "%lf", tempoGasto);
150     fprintf(pont_arq, "%s", "  - Conceitos: ");
151     fprintf(pont_arq, "%d \n\n", conceitos);
152   }
153
154
155   fclose(pont_arq);
156 }
157
158 // Entry point.
159 int main() {
160
161   printf("\nDigite a quantidade de objetos e atributos");
162   scanf("%d",qtdObjetos);
163   scanf("%d",combAtributos);
164
165   if(!init()) {
166     return -1;
167   }
168
169   printf("\n\nCriando Kernel...");
170   CreateKernel();
171
172   printf("\nCarregando contexto...");
173   CarregaContexto();
174
```

```
175   printf("\nIniciando Kernel...");
176   StartKernel();
177
178   int Conceitos = 0;
179   double start_time, end_time;
180
181   printf("\nExecutando Kernel...\n");
182   start_time = getCurrentTimestamp();
183
184   ExecutarKernel();
185
186   end_time += getCurrentTimestamp() - start_time;
187   printf("\n\nTempo de çãexecuo: %lf", end_time);
188
189   for(int z=0; z<combAtributos; z++)
190   {
191     Conceitos = Conceitos + retKernel[z];
192   }
193
194   Conceitos += 2;
195   printf("\n\nTotal de Conceitos: %d\n\n", Conceitos);
196
197   SalvarResultadoArquivo(end_time, Conceitos);
198
199   FinalizarKernel();
200
201   // Free
202   cleanup();
203
204   return 0;
205 }
206
207 bool init() {
208   cl_int status;
209
210   if(!setCwdToExeDir()) {
211     return false;
212   }
213
214   // Get the OpenCL platform. TODO: More than one platform can be found
215   cl_uint num_platforms;
216   status = clGetPlatformIDs(1, &platform, &num_platforms);
217   checkError(status, "Failed clGetPlatformIDs.");
218
219   // Query the available OpenCL devices.
220   scoped_array<cl_device_id> devices;
221   cl_uint num_devices;
```

```
222   devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));
223   device = devices[0]; // We'll just use the first device.
224
225   // Create the context.
226   context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
227   checkError(status, "Failed to create context");
228
229   // Create the command queue.
230   queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
          &status);
231   checkError(status, "Failed to create command queue");
232
233   // Create the program, using the name of aocx file
234   std::string binary_file = getBoardBinaryFile("analisarConceito", device);
235   program = createProgramFromBinary(context, binary_file.c_str(), &device,
          1);
236
237   // Build the program that was just created.
238   status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
239   checkError(status, "Failed to build program");
240
241   return true;
242 }
243
244 // Free the resources allocated during initialization
245 void cleanup() {
246   if(kernel) {
247     clReleaseKernel(kernel);
248   }
249   if(program) {
250     clReleaseProgram(program);
251   }
252   if(queue) {
253     clReleaseCommandQueue(queue);
254   }
255   if(context) {
256     clReleaseContext(context);
257   }
258 }
```

APÊNDICE C – ALGORITMO CPU

Código C.1 – Código da CPU

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <time.h>
6  #include <assert.h>
7  #include <omp.h>
8
9  int *dadosROM;
10 int qtdObjetos = 1000;
11 int combAtributos = 33554431;
12 int indice = 1;
13
14 void CarregaContexto(){
15
16   char Linha[25];
17   char *result;
18   char dir[200];
19   int aux, i = 0;
20
21   strcpy(dir, get_current_dir_name());
22   aux = strlen(dir);
23   dadosROM = (int*)malloc(sizeof(int) * qtdObjetos);
24
25   strcat(dir, "/baseTeste.txt");
26
27   FILE *arq = fopen(dir, "rt");
28
29   if (arq == NULL)
30   {
31       printf("Problemas na CRIACAO do arquivo\n");
32       return;
33   }
34
35   while (!feof(arq))
36   {
37     result = fgets(Linha, 25, arq);
38     if (result){
39       dadosROM[i] = atoi(Linha);
```

```
40      }
41      i++;
42    }
43    fclose(arq);
44  }
45
46  void SalvarResultadoArquivo(double tempoGasto, int conceitos){
47    FILE *pont_arq;
48
49    pont_arq = fopen("arquivoResultado.txt", "a");
50
51    if(pont_arq != NULL){
52      fprintf(pont_arq, "%s", "Testes de ");
53      fprintf(pont_arq, "%d", qtdObjetos);
54      fprintf(pont_arq, "%s", " Objetos e ");
55      fprintf(pont_arq, "%d", combAtributos);
56      fprintf(pont_arq, "%s", " Combinacoes\n");
57      fprintf(pont_arq, "%s", "Tempo gasto: ");
58      fprintf(pont_arq, "%lf", tempoGasto);
59      fprintf(pont_arq, "%s", "  - Conceitos: ");
60      fprintf(pont_arq, "%d \n\n", conceitos);
61    }
62    fclose(pont_arq);
63  }
64
65  bool ExecutarKernel(int indice){
66      int RX;
67      int i,j;
68      RX = combAtributos;
69      j=0;
70
71      ///BC IDA
72      for(i=0; i<qtdObjetos; i++) //Percorre todos os objetos da base (ómemria
            ROM)
73      {
74          if((indice & dadosROM[i]) == indice)
75          {
76            RX = RX & dadosROM[i];
77          }
78          else
79          {
80            j++;
81          }
82      }
83      if((RX == indice) && (j != qtdObjetos)){
84          return true;
85      }
```

```
86      return false;
87  }
88
89  int main() {
90    time_t Ticks[2];
91
92    printf("\nDigite a quantidade de objetos e atributos");
93    scanf("%d",qtdObjetos);
94    scanf("%d",combAtributos);
95
96    printf("\nCarregando contexto...");
97    CarregaContexto();
98
99    int ConceitosV[100], Conceitos;
100   for(int z=0; z<100; z++)
101   {
102      ConceitosV[z] = 0;
103   }
104
105   double start_time, end_time;
106
107   printf("\nExecutando Kernel...\n");
108
109   Ticks[0] = time(NULL);
110   int i;
111   #pragma omp parallel for private(i)
112   for (i = 1; i < combAtributos - 1; i++)
113   {
114     if(ExecutarKernel(i) == true){
115       ConceitosV[omp_get_thread_num()]++;
116     }
117   }
118   Ticks[1] = time(NULL);
119   end_time = difftime(Ticks[1], Ticks[0]);
120
121   for(int w=0; w<100; w++)
122   {
123      Conceitos += ConceitosV[w];
124   }
125   printf("\n\nTempo de çãexecuo: %lf", end_time);
126
127   Conceitos += 2;
128   printf("\n\nTotal de Conceitos: %d\n\n", Conceitos);
129
130   SalvarResultadoArquivo(end_time, Conceitos);
131
132
```

```
133     return 0;
134 }
```