Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em Ciência da Computação– PPGCC

Saulo Akamatu

Paralelização Híbrida do Algoritmo *Black Hole* em um System on Chip

São Carlos
2022

Saulo Akamatu

Paralelização Híbrida do Algoritmo *Black Hole* em um System on Chip

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Arquitetura de Computadores

Orientador: Prof. Dr. Emerson Carlos Pedrino

São Carlos
2022

Agradecimentos

À Deus. À minha esposa Eliana, pelo apoio incondicional na busca dos meus sonhos. Aos meus filhos, João Makoto e Maria Keiko, que mesmo na minha ausência, não entendendo o motivo me fizeram sentir sempre presente. Ao meu pai Durval (*in memorian*) e à minha mãe Joanita por fazerem de mim a pessoa que sou, espiritual e socialmente.

Ao meu orientador Prof. Dr. Emerson Carlos Pedrino, pela dedicação durante este período.

Ao programa de Pós-Graduação do Departamento de Ciências da Computação da UFSCar pela oportunidade concedida e aos professores que contribuíram no meu desenvolvimento durante este período.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) (código 001) pela bolsa concedida.

Ao secretário do programa, Ivan Rogério da Silva, que sempre deu o suporte quando precisei.

Aos amigos que fiz no departamento, Ricardo e William e, em especial Paulo e Denis, que contribuíram na minha formação como aluno e sempre me deram o suporte necessário.

Resumo

Black Hole (BH) é um algoritmo metaheurístico bioinspirado baseado na teoria da relatividade, em que uma massa suficientemente compacta pode deformar o espaço-tempo dando origem a um buraco negro. Deste fenômeno, nenhuma partícula ou radiação eletromagnética pode escapar. Tal abordagem é baseada no conceito de uma população de indivíduos (estrelas) representando soluções para um determinado problema computacional a ser otimizado. Na literatura, tal abordagem tem sido utilizada para resolver problemas de agrupamento, entre outros, por ser livre de parâmetros e simples de implementar. Devido a tais características, neste trabalho, foi proposta uma solução híbrida, em software/hardware, de paralelização do algoritmo BH, visando acelerar seu processamento em hardware. Para isso, foi utilizada uma metodologia que permite a qualquer usuário, mesmo não especialista, implementar aceleradores de hardware para problemas de otimização através de uma ferramenta de alto nível. Nesta implementação, foi utilizada uma plataforma System on Chip (SoC), contendo um chip Zynq da Xilinx, que possui dois núcleos ARM e um FPGA. O Algoritmo BH foi implementado primeiro em software e depois em hardware para avaliar o tempo de execução e validar esta abordagem. Além disso, neste trabalho, algoritmos de otimização mais simples e populares, como *Particle Swarm Optimization* (PSO), *Gravitational Search Algorithm* (GSA) e *Big Bang - Big Crunch* (BB-BC), juntamente com conjuntos de dados mais simples, foram utilizados para fins de uma comparação mais justa com o algoritmo BH, como realizado em outros trabalhos na literatura. Os resultados obtidos foram satisfatórios em termos de tempo de execução e qualidade, com ganho médio de *speedup* de 25 vezes em relação à mesma implementação em software.

Palavras-chave: Agrupamento de Dados. Algoritmo *Black Hole*. **SoC**. Hardware Híbrido.

Abstract

Black Hole (BH) is a bioinspired metaheuristic algorithm based on the theory of relativity in which a sufficiently compact mass can deform the space-time to form a black hole, where no particles or electromagnetic radiation can escape from it. Thus, such an approach is based on the concept of a population of individuals (stars) representing solutions for a given computational problem to be optimized. In the literature, such an approach has been used to solve clustering problems, among others, since it is parameter-free and simple to implement. In this work, due to such characteristics, a hybrid solution, in software/hardware, of parallelization of the BH algorithm is proposed, aiming at accelerating its processing in hardware through a methodology that allows any user, even a non-expert, implement hardware accelerators, for optimization problems, among others, through a high level tool. A SoC platform (Pynq) was used for this implementation, containing a Zynq chip from Xilinx, which has two ARM cores and an FPGA. The BH Algorithm was implemented in software first and then in hardware for runtime comparison purposes to validate this approach. Also, in this work, simpler and more popular optimization algorithms, such as Particle Swarm Optimization (PSO), Gravitational Search Algorithm (GSA), and Big Bang - Big Crunch (BB-BC), along with simpler databases, were used for comparison purposes, due to its ease of implementation and to keep a fairer comparison with BH as realized in other works in the literature. Therefore, the results obtained were satisfactory in terms of execution time and quality, with an average speedup of 25 times compared to the same implementation in software.

Keywords: *Data Clustering. Black Hole Algorithm.* **SoC**. *Hybrid Hardware.*

Lista de ilustrações

Figura 1 – Bloco Design do núcleo IP desenvolvido para o projeto. 36
Figura 2 – Solução candidata para o conjunto de dados *Iris*. Neste caso, o espaço de busca é de 4 dimensões. 38
Figura 3 – Estrutura da matriz ajustada para conjuntos de dados *Iris* e *Wine*. . . 39
Figura 4 – Estrutura de uma submatriz populacional e alinhamento com a submatriz do conjunto de dados. 40
Figura 5 – Diagrama de montagem das submatrizes. 45
Figura 6 – Mecanismo de cálculo da função objetiva. O conjunto de dados *Iris* foi usado, $k = 3$ e $N = 150$. Os operadores e expressões são da equação 1 46
Figura 7 – Diagrama geral da solução proposta. 47
Figura 8 – Algoritmo do Black Hole. Eq. 20 representa o movimento das estrelas em direção ao buraco negro; Eq. 19 representa o raio de ação do BH; f_{BH} é o valor da função objetivo do buraco negro; X_{BH} é a posição do buraco negro; e $t = 1, 2, ...$ *mximo nmero de iteraes*. A função *Accel* refere-se à parte acelerada pelo hardware, conforme explicado nesta seção. 48
Figura 9 – Testes dos Algoritmos BH, PSO, BB-BC e GSA, implementados em Matlab. Para o algoritmo PSO, $w = 0,51$, $d = 0,87$, $c1 = 1,37$ e $c2 = 1,01$ foram considerados; para o algoritmo BB-BC, $beta = 0,2$ e $alpha = 1$; e para o algoritmo GSA, $alpha = 20$ e $G_0 = 100$. 56
Figura 10 – Pseudo-código da função objetiva utilizada na amostra *BH SW* 58
Figura 11 – Tempo médio de execução para uma iteração com mais de 100 simulações independentes. 58
Figura 12 – Arquivo "*matmult.h*". Código em C++ da Implementação do acelerador em hardware. 68
Figura 13 – Arquivo "*matmult_accel.cpp*". Código em C++ da Implementação do acelerador em hardware. 69

Figura 14 – Código em C++ da Implementação do acelerador em hardware para a base *Iris* 70
Figura 15 – Código em C++ da Implementação do acelerador em hardware para a base *Wine*. 71
Figura 16 – Código em C++ da Implementação do acelerador em hardware para a base *Glass*. 72
Figura 17 – Código em C++ da Implementação do acelerador em hardware para a base *CMC*. 73
Figura 18 – Código da Montagem das Bases *Iris*, *Wine* e *Glass* 76
Figura 19 – Código da Montagem da Base *CMC*. 77
Figura 20 – Código em Python: Montagem das submatrizes da população para o conjunto *Iris* 80
Figura 21 – Código em Python: Montagem das submatrizes da população para o conjunto *Wine* 80
Figura 22 – Código em Python: Montagem das submatrizes da população para o conjunto *Glass* 81
Figura 23 – Código em Python: Montagem das submatrizes da população para o conjunto *CMC* 82
Figura 24 – Código em Python: Configuração do *Overlay* para as Bases *Iris*, *Wine*, *Glass* e *CMC* 84
Figura 25 – Código da função que executa o acelerador para os conjuntos *Iris*, *Wine* e *Glass* 84
Figura 26 – Código da função que executa o acelerador para o conjunto *CMC* . . . 85
Figura 27 – Definição dos parâmetros dos conjuntos de dados: *Iris*, *Wine*, *Glass* e *CMC*. O tamanho da solução candidata e limitações máximas e mínimas do espaço de busca também são definidos neste bloco de código. . 88
Figura 28 – Implementação do algoritmo BH para o conjunto *Iris* 89
Figura 29 – Implementação do algoritmo BH para o conjunto *Wine* 90
Figura 30 – Implementação do algoritmo BH para o conjunto *Glass* 91
Figura 31 – Implementação do algoritmo BH para o conjunto *CMC* 92

Lista de tabelas

Tabela 1 – Principais características das bases de dados. 37
Tabela 2 – Relação dos parâmetros usados neste trabalho (Número de soluções candidatas por submatriz e o tamanho da candidata e número de submatrizes por tamanho da população). 42
Tabela 3 – As menores somas das distâncias intragrupos obtidas por algoritmos implementados em Matlab e aplicados a diferentes conjuntos de dados. Para o algoritmo PSO foram considerados, $w = 0,51$, $d = 0,87$, $c1 = 1,37$ e $c2 = 1,01$; para o algoritmo BB-BC, $beta = 0,2$ e $alpha = 1$; e para o algoritmo GSA, $alpha = 20$ e $G_0 = 100$. 51
Tabela 4 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Iris*. 52
Tabela 5 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Wine*. 52
Tabela 6 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Glass*. 52
Tabela 7 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *CMC*. 53
Tabela 8 – A taxa de erro (%) dos algoritmos testados. **BH*** representa os valores extraídos da literatura (HATAMLOU, 2013). 54
Tabela 9 – Classificação média obtida pelo teste de *Friedman* utilizando os parâmetros: "valor médio da soma das distâncias intragrupos"e "taxas de erro". 55
Tabela 10 – Resultados dos testes de *Friedman* e *Iman-Davenport* com base na soma das distâncias intragrupos 55
Tabela 11 – Tempo médio de execução para uma iteração com mais de 100 simulações independentes. 57

Lista de siglas

AMBA *Advanced Microcontroller Bus Architecture*

ARM *Advanced RISC Machine*

AXI *Advanced eXtensible Interface*

BB-BC *Big Bang - Big Crunch*

BH *Black Hole*

CMC *Contraceptive Method Choice*

CMOS *Complementary Metal Oxide Semiconductor*

CPU *Central Process Unit*

DMA Direct Memory Access

DRAM *Dynamic Random Access Memory*

FPGA *Field Programmable Gate Array*

GSA *Gravitational Search Algorithm*

HDL *Hardware Descripption Language*

HLS *High-Level Synthesis*

IP Intellectual Property

MMIO *Memory-Mapped Input / Output*

NP *Non-Deterministic Polynomial time*

PL programação lógica

PSO *Particle Swarm Optimization*

SP sistema de processamento

SoC System on Chip

Tcl *Tool Command Language*

VHDL *Very High Speed Integrated Circuit Hardware Description Language*

Sumário

1 INTRODUÇÃO 17
1.1 Objetivos 20
1.2 Organização do Trabalho 20

2 FUNDAMENTOS TEÓRICOS 21
2.1 Agrupamento de Dados 21
2.2 Algoritmos Heurísticos 22
2.2.1 Algoritmo de Otimização de Partículas (PSO) 23
2.2.2 Algoritmo *Big Bang - Big Crunch* (BB-BC) 24
2.2.3 Algoritmo de Busca Gravitacional (GSA) 25
2.3 Algoritmo *Black Hole* (BH) 28
2.4 Arquitetura Zynq SOC 29
2.5 Trabalhos Correlatos 31

3 METODOLOGIA E DESCRIÇÃO DA IMPLEMENTAÇÃO 35
3.1 Desenvolvimento do Acelerador em Hardware 35
3.2 Estrutura de Dados para Agrupamento dos Dados 37
3.3 Implementação do Algoritmo *Black Hole* 42

4 RESULTADOS E DISCUSSÕES 49
4.1 Testes dos Algoritmos BH, PSO, BB-BC e GSA para Agrupamento de Dados 49
4.1.1 Estudo da soma das distâncias euclidianas intragrupos 50
4.1.2 Estudo da porcentagem de erro de agrupamento de dados 53
4.1.3 Estudo do tempo de execução dos algoritmos 55
4.2 Estudo do tempo de execução do algoritmo BH implementado na Pynq-Z1 57

5 CONCLUSÃO 59

REFERÊNCIAS 61

APÊNDICE A CÓDIGO UTILIZADO PARA O DESENVOLVIMENTO DO ACELERADOR EM HARDWARE NO SOFTWARE VIVADO HLS 67

APÊNDICE B CÓDIGO EM PYTHON: MONTAGEM DAS SUBMATRIZES DOS CONJUNTOS DE DADOS *IRIS*, *WINE*, *GLASS* E *CMC* 75

APÊNDICE C CÓDIGO EM PYTHON: MONTAGEM DAS SUBMATRIZES DA POPULAÇÃO 79

APÊNDICE D IMPLEMENTAÇÃO DO *OVERLAY* EM PYTHON 83

APÊNDICE E CÓDIGO EM PYTHON: IMPLEMENTAÇÃO DO ALGORITMO *BLACK HOLE* 87

Capítulo 1

Introdução

O avanço e o desenvolvimento de novas tecnologias, bem como de aplicações modernas em processos industriais e comerciais, podem ser baseados no uso de diferentes metaheurísticas, tendo-se em vista resolver problemas complexos de otimização no mundo real (DITZLER et al., 2015). Processamento de imagens, controle industrial e análise de dados são exemplos de aplicações que exigem alto desempenho computacional. Na área de inteligência artificial, sistemas inteligentes embarcados representam um campo de estudo altamente correlacionado ao surgimento de aplicações modernas (STORNAIUOLO; SANTAMBROGIO; SCIUTO, 2018) e seus desafios (TREFZER; TYRRELL, 2015; BAE, 2021; SZE et al., 2017; BASU et al., 2018; SMITH, 2010; MOLANES et al., 2018).

Diversas áreas (e subáreas) da computação como Sistemas Neuromórficos, Hardware Evolucionário, Inteligência Artificial, Aprendizado de Máquina, Redes Neurais e suas diferentes abordagens interessam-se por sistemas embarcados inteligentes.

Estudos que abordaram fundamentos, progressos e tendências nessa área de pesquisa ressaltaram os desafios e interesses diante dos avanços e inovações em tecnologias voltadas para soluções de hardware (SMITH, 2010; BASU et al., 2018). Um exemplo é o estudo de sistemas neuromórficos, com implementações em nível de chip, os quais buscam simular um sistema neural (LIU et al., 2014). Assim, os estudos destacam que os novos métodos e suas abordagens de implementações com novos algoritmos para implementação em hardware e técnicas de aprendizado não supervisionado têm justificado o desenvolvimento de certos tipos de sistemas de sensoriamento remoto em modelagem neural, interfaces cérebro-máquina, robótica etc. Arquiteturas, circuitos e dispositivos são necessários para permitir a inteligência adaptativa de sistemas neuromórficos, especialmente em sistemas embarcados com severas restrições ao minimizar efeito de potência e densidade do chip (ALIPPI et al., 2009). Outras áreas de pesquisa também compartilharam esses desafios

e tendências relacionadas, conforme apontam Sze et al. (2017), que estudaram sobre os desafios e oportunidades em soluções de hardware para aprendizado de máquina. Os autores destacam a importância de explorar projetos de hardware em vários níveis para equilibrar precisão, latência, requisitos de custo e, especialmente, eficiência energética. Por exemplo, técnicas de paralelismo como aceleradores de hardware são propostas para otimizar problemas representados de forma matricial e desenvolver um fluxo de dados eficiente, reutilizando dados em baixo nível, reduzindo, com isso, os custos de acesso à memória e, consequentemente, os custos de energia. Outra ênfase igualmente importante é a percepção na escolha do algoritmo a ser implementado em hardware, evitando implicar maior complexidade ao sistema.

Tais questões têm despertado grande interesse e, em muitos aspectos, já estão endereçadas. Bae (2021), nesse tópico, discute trabalhos relacionados a novas tecnologias de memória como os CMOS, que buscam otimizar o consumo de energia, a área de uso em nível de nanotecnologia e a minimização de latência; na parte de processamento, arquiteturas cada vez mais poderosas são estudadas; arquiteturas com multiprocessadores e técnicas de paralelismo sem aumento de *clock*, evitando, consequentemente, a dissipação de calor, o que prejudica o desempenho do sistema em geral (ALIPPI et al., 2009).

Além disso, a parte de conectividade abrange estudos para superar limitações de largura de banda, potência e velocidade. No entanto, toda essa evolução tecnológica que envolve projetos de hardware está relacionada às limitações dos dispositivos eletrônicos existentes. Outro desafio no projeto de hardware é a demanda por um vasto conhecimento multidisciplinar, muitas vezes de obtenção só possível por meio de grupo de especialistas qualificados e experientes, exigindo-se alto investimento de mão de obra qualificada.

Por esse motivo, o custo e o tempo de implementação são também limitações de projetos em hardware. Salvador (SALVADOR, 2016) analisou algumas formas de contornar esses pontos: reutilizar métodos, abordagens e aplicações desenvolvidas em computação reconfigurável com suporte para linguagens de programação de alto nível; automação de projetos digitais e suporte para simulações e validações em tempo real em vários níveis do sistema em desenvolvimento. Dessa forma, o desenvolvimento de projetos em hardware se torna mais simples e flexível, provendo maior produtividade dos desenvolvedores.

Quanto às aplicações em sistemas embarcados, vale citar o estudo realizado por Honda et al. (HONDA; WEI; AMANO, 2019), que implementaram uma solução de detecção de faixa de pista em uma Pynq-Z1 ("PYNQ", 2022). A detecção de faixa é um componente crítico para carros autônomos e para a visão computacional em geral. Esse conceito é usado para descrever o caminho para carros autônomos e evitar o risco desses entrar em outra faixa. Os autores notaram os benefícios de utilizar a linguagem de programação Python de forma nativa na SoC, linguagem destacada por ter uma riqueza de bibliotecas atraentes, como a Scikit-learn, a Django etc.

O alto processamento computacional foi mitigado usando uma biblioteca de eficácia

rápida, por meio da FPGA contida na Pynq, que reduziu o tempo de resposta da solução e o custo de desenvolvimento do projeto. Já Wang et al. (WANG et al., 2019) propuseram um acelerador de hardware implementado em uma Pynq para otimizar o algoritmo K-média baseado no conceito de desigualdade triangular, de forma a lidar com grandes conjuntos de dados de várias dimensões, sendo possível concluir que a implementação proposta superou consistentemente um padrão otimizado baseado em CPU, aumentando a velocidade e melhorando a eficiência energética.

Conforme abordagem feita por Bouvier et al. (BOUVIER et al., 2019)), há ainda muito a melhorar do ponto de vista de algoritmos implementados de forma híbrida em hardware como em redes neurais e suas diferentes abordagens, sistemas neuromórficos e aplicações de aprendizado profundo. Isso sugere que novos algoritmos (e mesmo os tradicionais) devem ser sempre explorados.

Nesse cenário, para obter resultados satisfatórios, é interessante utilizar algoritmos sem dependências de parametrização. Alguns estudos, como (HATAMLOU, 2013; KUMAR; DATTA; SINGH, 2015; TSAI; HSIEH; CHIANG, 2015) abordaram o algoritmo *Black Hole* (BH), concluindo ser este um algoritmo simples, fácil, sem parâmetros e potencialmente poderoso para agrupamentos. Kumar et al. (2015) enfatizaram outros pontos além dos já citados, reconhecendo que o algoritmo BH possui uma abordagem semelhante a outros algoritmos; o algoritmo BH é flexível quanto a ser modificado e combinado com outros algoritmos, o que despertou interesse na área de redes neurais.

Diante do exposto, neste trabalho, é apresentada uma solução de hardware híbrido do algoritmo BH mediante uma abordagem que permite qualquer programador, mesmo não especialista em hardware, implementar um hardware dedicado. O BH foi implementado em uma Pynq-Z1 para explorar o processamento paralelo e foi testado com o propósito de resolver o agrupamento de quatro conjuntos de dados simples comumente usados na literatura. Além disso, outros algoritmos de otimização mais simples e populares foram explorados melhor em software para fins de agrupamento de dados. São eles: *Particle Swarm Optimization* (PSO), *Gravitational Search Algorithm* (GSA) e *Big Bang - Big Crunch* (BB-BC). Este estudo foi importante na avaliação do parâmetro tempo de execução e, ainda, para validar a implementação proposta.

Cabe ressaltar, ainda, que os resultados deste estudo contribuirão significativamente para trabalhos futuros na área, uma vez que fornecem todos os parâmetros utilizados para a implementação como tamanho da população e número de iterações para alcançar um valor ótimo global, os quais não foram relatados em estudos semelhantes (HATAMLOU; ABDULLAH; HATAMLOU, 2011; HATAMLOU, 2013; NIKBAKHT; MIRVAZIRI, 2015). Esses parâmetros são fundamentais para entender completamente os resultados considerados ótimos.

1.1 Objetivos

O objetivo geral deste trabalho foi a hibridização e paralelização do algoritmo BH em um SoC para resolver o agrupamento de alguns conjuntos de dados. Para o desenvolvimento desta proposta, os objetivos específicos foram:

- ❏ replicar os algoritmos BH, PSO, GSA e BB-BC no software Matlab (HIGHAM DESMOND J E HIGHAM,);
- ❏ realizar experimentos para ajustes de parâmetros de tamanho de população, número de iterações e tempo de execução;
- ❏ desenvolver e implementar quatro aceleradores em hardware, um para cada conjunto de dados, nos softwares Vivado HLS (WINTERSTEIN; BAYLISS; CONSTANTINIDES, 2013) e Vivado Suite Design ("VIVADO", 2020);
- ❏ implementar o algoritmo BH, de forma híbrida na Pynq-Z1, utilizando os aceleradores em hardware desenvolvidos;
- ❏ implementar o algoritmo BH na Pynq-Z1, sem utilização dos aceleradores.

1.2 Organização do Trabalho

A presente dissertação está organizada em cinco capítulos. No Capítulo 1 foram apresentados o contexto e os conceitos teóricos básicos relacionados ao tema proposto. No Capítulo 2, são apresentados os fundamentos teóricos das técnicas de agrupamento de dados utilizadas neste trabalho, dos algoritmos BH, PSO, GSA, BB-BC e da plataforma Pynq-Z1, enfocando os pressupostos básicos que fundamentam a abordagem proposta, delineando também o algoritmo BH. No Capítulo 3, o framework montado para o desenvolvimento e implementação do trabalho foi detalhado, enquanto os testes e resultados foram descritos no Capítulo 4. O Capítulo 5 apresenta as considerações finais do trabalho, ressaltando importância da pesquisa e as conclusões do estudo. Por fim, são apresentadas as referências utilizadas para a escrita desta dissertação, seguida dos Apêndices.

Capítulo 2

Fundamentos Teóricos

2.1 Agrupamento de Dados

A tarefa de agrupamento de dados consiste em identificar a existência de diferentes grupos dentro de um determinado conjunto de dados. Uma forma de realizar esta tarefa é por meio da aprendizagem não-supervisionada, na qual não há suposição inicial a respeito dos dados existentes, nem predefinição de classes ou treinamento com classes rotuladas. Nesse caso, o objetivo é construir agrupamentos somente a partir das semelhanças entre os dados, extraídas das informações presentes nas variáveis ou atributos (HATAMLOU; ABDULLAH; HATAMLOU, 2011).

A semelhança entre os dados pode ser medida por meio da distância Euclidiana entre o objeto e o centro de cada grupo. Dessa maneira, associa-se o objeto ao grupo cujo centro se encontra mais próximo e, então, os objetos de um mesmo grupo devem possuir características similares entre si e dissimilares em relação aos objetos dos demais grupos (JAIN, 2010).

O problema é constituído de N objetos, sendo que cada objeto é associado somente a um dos K grupos. A soma das distâncias euclidianas entre todos os objetos e seus respectivos centroides, ao quadrado, está representada na equação 1, em que $\|O_i - Z_j\|$ é a distância euclidiana entre o objeto O_i e o centroide Z_j. O fator peso w_{ij} é associado a cada objeto i e ao grupo j, onde o objeto assumirá o valor 1 ou 0. Se o objeto i estiver associado ao grupo j, então w_{ij} será igual a 1. Caso contrário, seu valor será zero. Dessa forma, a menor soma das distâncias intragrupos significa haver maior similaridade dos objetos intragrupos e maior dissimilaridade dos objetos intergrupos.

$$F(O, Z) = \sum_{i=1}^{N} \sum_{j=1}^{K} w_{ij} \|O_i - Z_j\|^2 \quad (1)$$

De forma geral, os algoritmos tentam aprimorar a população de forma iterativa. A cada iteração, é calculada a função objetiva de cada solução candidata da população, a qual é representada pela equação 1. O candidato que apresentar a menor soma das distâncias intragrupos representa os centroides mais bem posicionados no espaço de busca. Inspirados cada um em sua teoria, os algoritmos deste estudo tentam mimetizar processos naturais que resultem em uma busca otimizada no espaço. (repensar)

2.2 Algoritmos Heurísticos

Neste estudo, faz-se a busca da solução partindo de uma população inicial de candidatos a uma solução ótima. Cada candidato representará k grupos (centroides), sendo que cada centroide será representado na mesma dimensão dos objetos da base de dados a ser analisado. A busca exploratória é feita pela geração aleatória dos candidatos no espaço de busca delimitado.

O posicionamento inadequado dos centroides de um candidato qualquer pode acarretar baixa performance do algoritmo, obtendo-se, consequentemente, um resultado não esperado como uma convergência prematura em um ponto mínimo local (JAIN, 2010). No entanto, esses problemas geralmente não são lineares e são restritos ao próprio problema a ser otimizado. Exemplos de restrições geralmente são requisitos de tempo reduzido e alta dimensionalidade para encontrar a solução ótima global.

Essa área de estudo fomentou vários campos de pesquisa. O algoritmo k-médias, por exemplo, teve sua primeira publicação em 1955 e ainda é um dos algoritmos de agrupamento de dados mais populares (JAIN, 2010). Apesar de antigo, o algoritmo é amplamente utilizado, pois um dos desafios é a dificuldade de obter um algoritmo de agrupamento de dados de propósito geral que satisfaça qualquer problema de agrupamento. Muitos estudos publicados demonstram que, mesmo utilizando diferentes abordagens dentro do conceito do k-médias, o desempenho do algoritmo está relacionado aos resultados esperados da aplicação em questão, da análise e da manipulação do estudo.

Não se pode concluir que exista uma abordagem única, na busca de uma solução, que tenha o melhor resultado diante de um problema de otimização de propósito geral (JAIN, 2010; SAEGUSA; MARUYAMA, 2007; FILHO et al., 2003). Desde então, diversos algoritmos distintos de agrupamento de dados foram definidos. Dentre as áreas em que algoritmos de agrupamento de dados são empregados, incluem-se a análise de dados, a aprendizagem de máquina, a análise de imagens, a mineração de texto, a bioinformática etc.

Os algoritmos inspirados na natureza, nas leis da física e nos fenômenos do universo possuem a habilidade de realizar buscas em grandes espaços de soluções, cobrindo um grande subconjunto do espaço de busca de forma eficaz. Problemas de otimização NP-completos, como é o caso da tarefa de agrupamento de dados, são cobertos de forma eficaz e proveem boas soluções em tempo razoável. Devido a isso, os algoritmos PSO, BB-BC, GSA e o algoritmo BH foram utilizados neste trabalho.

2.2.1 Algoritmo de Otimização de Partículas (PSO)

O algoritmo PSO (KENNEDY; EBERHART, 1995) é inspirado no comportamento de um enxame de partículas, como o voo de um bando de pássaros ou o nado sincronizado de um cardume. Cada partícula (candidato) possui um vetor de posição e um vetor de velocidade que calcula a nova posição da partícula a cada iteração. A velocidade de cada partícula é influenciada pela própria experiência da partícula, bem como a experiência de seus vizinhos. Existem duas variantes de vizinhança no PSO: local e global. A cada iteração é feita a movimentação das soluções candidatas no espaço de busca, usando-se os melhores locais encontrados pela própria partícula (variante local) e o melhor local encontrado pelo bando (variante global), que são atualizados conforme novos e melhores locais encontrados. As variantes estão presentes na função de velocidade para manipular a distância entre partículas que causam o efeito de sincronização do bando. Em outros termos, a função velocidade busca manter uma distância ideal entre as partículas e seus vizinhos. O algoritmo PSO atinge a convergência quando todas as partículas se encontram muito próximas uma da outra, com seus valores de aptidão significativamente próximos ou quando uma quantidade máxima de iterações, estabelecidas previamente, for atingida.

A população consiste em partículas e, para cada iteração, uma função objetiva f é usada para medir a aptidão de cada uma destas na população. A posição de cada partícula i é, então, atualizada, sendo influenciada por três termos: a velocidade da partícula desde a última iteração, a diferença entre a melhor posição conhecida da partícula e a posição corrente desta, e a diferença entre a melhor posição dentre todas as partículas do bando e sua posição corrente. Os dois últimos termos são multiplicados por um número aleatório, distribuído uniformemente no intervalo (0,1), pertencente aos números reais. Dessa forma, a influência de cada termo varia aleatoriamente, existindo um coeficiente de aceleração para dimensionar e equilibrar a influência de cada termo. A melhor posição obtida por cada partícula é armazenada no vetor p_i, conhecido como *pbest*, enquanto a melhor posição alcançada dentre todas as partículas na população é armazenada no vetor conhecido como *gbest*. O vetor de velocidade v_i para cada partícula é atualizado conforme a equação 2:

$$v_i{}^{t+1} = wv_i^t + c_1 r_1.(p_i^t - x_i^t) + c_2 r_2.(p_g^t - x_i^t) \tag{2}$$

Na equação, w, c_1 e c_2 são parâmetros positivos e r_1 e r_2 são números aleatórios distribuídos uniformemente no intervalo (0,1), pertencente aos números reais. O coeficiente de inércia w é usado para manter as partículas se movendo na mesma direção em que eles estão viajando, e tipicamente assume um valor aleatório entre (0,1). O termo c_1 é chamado de aceleração cognitiva, e c_2 é chamado de aceleração social. Esses dois valores equilibram a influência entre o melhor desempenho da partícula (cognitiva) e o da população (social). A velocidade é restringida entre os parâmetros $Vmin$ e $Vmax$ para limitar a mudança máxima de posição:

$$v_i{}^{t+1} = \begin{cases} v_{Max} & \text{se } v_i{}^{t+1} > V_{max} \\ v_{Min} & \text{se } v_i{}^{t+1} < V_{min} \end{cases} \tag{3}$$

A posição de cada partícula é atualizada usando o novo vetor de velocidade:

$$x_i{}^{t+1} = x_i{}^{t} + v_i{}^{t+1} \tag{4}$$

A posição em cada dimensão é limitada entre os parâmetros X_{min} e X_{max}:

$$x_i{}^{t+1} = \begin{cases} x_{Max} & \text{se } x_i{}^{t+1} > X_{max} \\ x_{Min} & \text{se } x_i{}^{t+1} < X_{min} \end{cases} \tag{5}$$

A solução atinge a convergência quando todas as partículas se encontram maximamente próximas umas às outra, com seus valores de aptidão próximos de um valor ótimo global, de forma significativa, diante do tamanho de valor de erro aceitável.

2.2.2 Algoritmo *Big Bang - Big Crunch* (BB-BC)

Com base nas teorias da evolução do universo envolvendo as fases *Big Bang* e *Big Crunch*, um novo algoritmo evolutivo foi desenvolvido, sendo chamado de algoritmo BB-BC (EROL; EKSIN, 2006).

A principal característica da fase do *Big Bang* é a dissipação de energia que produz desordem e aleatoriedade no espaço quanto à movimentação das partículas. Por outro lado, ordenar as partículas que estão distribuídas aleatoriamente é característica da fase *Big Crunch*, na qual são reduzidos os pontos em um único ponto representativo, utilizando-se uma abordagem de centro de massa, ou de menor custo, dentre todos os pontos. A cada iteração ocorre um *Big Bang* e um *Big Crunch*.

Após uma série de *Big Bangs* e *Big Crunches*, a distribuição aleatória no espaço de pesquisa durante o *Big Bang* fica menor sobre o ponto médio calculado durante o *Big*

Crunch e, então, o ritmo converge para uma solução. Sucessivas explosões e contrações, mimetizando a fase *Big Bang* e a fase *Big Crunch*, são realizadas repetidamente, até que um critério de término seja atendido.

O algoritmo BB-BC foi aplicado com sucesso, por exemplo, para estimar parâmetros de sistemas estruturais, controlar sistemas não lineares, dentre outras aplicações (KAVEH; TALATAHARI, 2010; KAVEH; TALATAHARI, 2009; KUMBASAR et al., 2011; TANG et al., 2010).

Especificamente, a fase *Big Crunch* funciona como um operador de convergência com muitas entradas (soluções candidatas), mas apenas uma saída (centro de massa). A fórmula da equação 6 é usada para calcular o ponto que representa o centro de massa (x^c):

$$x^c = \frac{\sum_{i=1}^{N} \frac{1}{f^i} x^i}{\sum_{i=1}^{N} \frac{1}{f^i}} \tag{6}$$

onde, x^i é a solução gerada dentro do espaço de n dimensões, f^i é o valor da função de avaliação da solução e N é a quantidade de candidatos à solução que a fase *Big Bang* terá. Depois da fase *Big Crunch*, novamente ocorre a fase *Big Bang* e uma nova geração de candidatos é produzida, agora com conhecimento prévio do centro de massa da população anterior, distribuindo a nova geração ao redor do centro de massa de forma uniforme em todas as direções. Quando o número de iterações cresce, o desvio padrão da distribuição normal da função de avaliação diminui o que pode ser notado na equação 7 que representa a fase *Big Bang*.

$$x^i_{new} = x^c + \frac{lr}{k} \quad i = 1, 2, ..., N \tag{7}$$

onde r é um número aleatório distribuído uniformemente no intervalo (0,1), pertencente aos números reais, com mudanças para cada candidato. l é um parâmetro para limitar o tamanho do espaço de busca e k é o passo da iteração.

2.2.3 Algoritmo de Busca Gravitacional (GSA)

O GSA é um algoritmo heurístico que foi primeiramente desenvolvido por Esmat Rashedi (RASHEDI; NEZAMABADI-POUR; SARYAZDI, 2009), com base na teoria da Gravitação de Newton (RASHEDI; RASHEDI; NEZAMABADI-POUR, 2018; MAHDAD; SRAIRI, 2015; JIANG; WANG; JI, 2014).

Na tentativa de imitar a influência das forças gravitacionais que reagem entre partículas de massa, o GSA considera que cada partícula possui uma massa e uma posição no espaço de busca. Cada partícula do universo atrai todas as outras partículas com uma força que é diretamente proporcional ao produto de suas massas e inversamente

proporcional ao quadrado da distância entre elas (HALLIDAY; RESNICK; WALKER, 1993).

De forma análoga, uma população de partículas é gerada num espaço aleatório, no qual quanto melhor localizada estiver uma partícula, maior será sua massa. Mimetizando a força da gravidade, busca-se um movimento geral de todas as partículas em direção às partículas com maior massa. As massas mais pesadas se movem mais lentamente em relação às partículas de menor massa e são consideradas boas soluções.

A cada iteração, o GSA, baseado nas fórmulas de forças atuantes entre as partículas, obtém o cálculo de aceleração, velocidade e deslocamento de cada partícula no espaço de busca. Uma constante gravitacional é reduzida em função do tempo, controlada pelo número de iterações e tende a zero na última iteração. Com isso, ganha-se maior acuracidade na pesquisa no espaço (ELAZAB et al., 2018). Porém, quanto maior a precisão, maior o custo computacional.

Para compensar essa demanda computacional, a população inicial de partículas é reduzida linearmente ao longo do tempo, considerando-se que apenas um conjunto de partículas com maior massa aplique sua força uns aos outros. A redução é feita até que apenas o melhor candidato atue nas demais partículas. Por fim, é verificado se o critério de parada foi atingido. Se sim, a posição da partícula com melhor valor de avaliação é retornada. Caso contrário, o algoritmo refaz todas as etapas anteriores, considerando a população corrente como uma nova população.

No GSA, cada solução no espaço de busca é chamada de agente, e cada agente reage com outros agentes em razão da força da gravidade. O desempenho de cada agente é determinado por sua massa e, portanto, cada agente é representado por uma partícula com características específicas. Devido à força da gravidade, impõe-se um movimento geral de todas as partículas em direção às partículas com maior massa. As massas mais pesadas movem-se mais lentamente em relação às partículas de menor massa e são consideradas boas soluções. O GSA explora a solução ideal, ajustando a massa gravitacional e a massa inercial. Para explicar o mecanismo do GSA, são considerados N agentes para descrever um sistema (RASHEDI; NEZAMABADI-POUR; SARYAZDI, 2009). A localização de cada agente em um espaço de pesquisa de n dimensões é definida por:

$$X_i(t) = (x_i{}^1(t), x_i{}^2(t), ..., x_i{}^d(t), ..., x_i{}^n(t)) \quad for \quad i = 1, 2, 3..., N. \tag{8}$$

onde $x_i{}^d(t)$ representa a posição do agente i na dimensão n no tempo t. A equação 9 representa a fórmula da constante gravitacional G:

$$G(t) = G_o e^{-\alpha t/T} \tag{9}$$

onde G_o e α iniciam com os valores 100 e 20 respectivamente. De acordo com Elazab et al.,

a cada cálculo da equação 9, conforme o t é incrementado, o valor é diminuído obtendo uma melhor acurácia da busca (ELAZAB et al., 2018). As massas gravitacionais e de inércia são simplesmente calculadas pela avaliação da aptidão. Uma massa mais pesada significa um agente mais eficiente, ou seja, melhores agentes têm atrações mais altas e andam mais devagar. Assumindo a igualdade entre a massa gravitacional e a massa de inércia, os valores das massas são calculadas usando o mapa de soluções aptas a resolver o problema. A gravidade e a inércia são atualizadas pelas equações 10, 11 e 12:

$$M_{ai} = M_{pi} = M_{ii} = M_i \quad i = 1, 2, ..., N, \tag{10}$$

$$m_i(t) = \frac{fit_i(t) - worst(t)}{best(t) - worst(t)}, \tag{11}$$

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}, \tag{12}$$

onde $fit_i(t)$ representa o valor de aptidão do agente i no momento t, e $worst(t)$ e $best(t)$ são adequados para um problema de minimização ou maximização. T é o número total de iterações. É importante determinar a força gravitacional entres agentes para cada tempo i. Se assumirmos que o agente j atua sobre o agente i, então a força gravitacional entre eles pode ser calculada pela seguinte equação:

$$F_{ij}^d(t) = G(t)\frac{M_{pi}(t) \times M_{ai}(t)}{R_{ij} + \epsilon}(x_j^d(t) - x_i^d(t)), \tag{13}$$

onde $M_{ai}(t)$ é a medida de força gravitacional atuante sobre um determinado agente i. A força do campo gravitacional de um agente i com menor massa gravitacional ativa, é mais fraca do que um agente de maior massa. $M_{pi}(t)$ é a medida de força de interação de um agente dentro do campo gravitacional. Em um mesmo campo gravitacional, um agente com menor massa gravitacional passiva sofre uma menor força do que um agente com maior massa gravitacional passiva. R_{ij} é a distância Euclidiana entre o agente i e o agente j, e ϵ é uma constante para evitar uma divisão por zero (RASHEDI; RASHEDI; NEZAMABADI-POUR, 2018). A somatória de todas as forças atuantes no agente i, é calculada conforme equação 14:

$$F_i^d(t) = \sum_{j \in best\ sol., j \neq i} rand_j F_{ij}^d(t) \tag{14}$$

onde $rand_j$ retorna um único número aleatório distribuído uniformemente no intervalo (0,1), pertencente aos números reais. Então, a aceleração $\alpha_i^d(t)$ é especificada a seguir:

$$\alpha_i^d(t) = \frac{F_i^d(t)}{M_{ii}(t)}, \tag{15}$$

onde $M_{ii}(t)$ é a inércia da massa do agente i, ou seja, é a medida de resistência que um agente i sofre para se movimentar. Um agente com maior massa inercial se movimenta mais lentamente do que um agente com menor massa inercial. Durante a busca no espaço, a velocidade e posição de cada agente é atualizado conforme as equações 16 e 17, respectivamente:

$$V_i^d(t+1) = rand_i \times V_i^d(t) + \alpha_i^d(t), \tag{16}$$

$$x_i^d(t+1) = x_i^d(t) + V_i^d(t+1), \tag{17}$$

onde $rand_i$ retorna um único número aleatório distribuído uniformemente no intervalo (0,1), pertencente aos números reais. Resumidamente, o GSA começa gerando uma população inicial de agentes de forma aleatória. Calcula-se a função de aptidão de cada agente e, então, atualizam-se informações do melhor e pior valor obtido de cada população. Em seguida, a massa inercial e aceleração de cada agente são calculadas, e atualizam a velocidade e posição de cada agente. Por fim, é verifica-se o critério de parada foi atingido. Se sim, a posição do agente com melhor valor de aptidão é retornado. Caso contrário, o algoritmo refaz todas as etapas anteriores, considerando a população corrente como uma nova população.

2.3 Algoritmo *Black Hole* (BH)

Este algoritmo tem inspiração no fenômeno do buraco negro para buscar uma solução quase ótima e, por isso, é denominado *Black Hole Algorithm*. O buraco negro possui um centro de força gravitacional capaz de colapsar a matéria de tudo que estiver no seu raio de ação como as estrelas e até mesmo a luz (GIACCONI, 2001). Essa região em que o buraco negro exerce a força gravitacional é conhecida como horizonte de eventos. A equação 18, *Schwarzschild Radius*, fornece esse raio de ação (HATAMLOU, 2013).

$$R = \frac{2GM}{c^2} \tag{18}$$

Nessa fórmula, G é a constante gravitacional, M é a massa do buraco negro e c é a velocidade da luz. O conjunto inicial de candidatos à solução, denominados estrelas, é gerado de forma aleatória no espaço de busca. Dessa forma, o algoritmo tenta explorar ao máximo um espaço desconhecido, que pode ser infinito, em busca da solução, sabendo que em seu raio de ação tudo será colapsado. A solução buscada identifica qual estrela da população será considerada buraco negro; vale entender, cada estrela é um candidato a ser um buraco negro. De acordo com Hatamlou (2013), utilizando a fórmula do horizonte

de eventos, equação 19, e a função objetiva, equação 1, pode-se verificar se a estrela será um buraco negro, ou se será colapsada. Assim, temos:

$$R = \frac{f_{BH}}{\sum_{i=1}^{N} f_i} \tag{19}$$

$$x_i(t+1) = x_i(t) + rand \times (x_{BH} - x_i(t)) \tag{20}$$

onde f_{BH} é o valor de aptidão do buraco negro, f_i é o valor de aptidão da i-ésima estrela e N é o número de estrelas. Quando a distância entre a estrela e o buraco negro atual for menor do que R, a estrela é colapsada e uma nova estrela é gerada em uma localização aleatória no espaço de busca. Este mecanismo permite uma busca mais exploratória, já que as novas estrelas serão geradas de forma independente de qualquer inclinação de dados.

Ainda, N é o número de estrelas (soluções candidatas), $x_i(t)$ e $x_i(t+1)$ são as posições da $i - sima$ estrela nas iterações t e $t+1$, respectivamente. O X_{BH} é a posição do buraco negro no espaço de busca investigado. *Rand* retorna um único número aleatório distribuído uniformemente no intervalo (0,1), pertencentes aos números reais. As estrelas avançam no espaço de busca a um passo controlado, sem um deslocamento maior que a distância entre a estrela e o buraco negro. Este mecanismo (Eq. 20) representa uma varredura mais minuciosa e criteriosa do algoritmo BH, pois o avanço no espaço de busca acontece de forma orientada. Ao se deslocar em direção ao buraco negro (X_{BH}), uma estrela pode chegar a uma posição com custo melhor do que o buraco negro. Se isto ocorrer, o buraco negro muda para a posição dessa estrela e vice-versa. Assim, o algoritmo BH vai manter o X_{BH} na nova posição, e então, as estrelas começam a se mover em direção a ele. O critério de parada ocorre após atingir um número máximo de iterações t ou um f_{BH} satisfatório for atendido.

2.4 Arquitetura Zynq SOC

Desde a introdução do Virtex II Pro com processadores PowerPC, há quase duas décadas (LYSAGHT, 2003), os FPGAs evoluíram para dispositivos de processamento heterogêneo mais sofisticados, com subsistemas de processadores baseados em ARM completos, unidades de ponto flutuante e controladores de memória (GU; HAO; YANG, 2018).

Estudos recentes (WANG et al., 2019; JANSSEN; ZIMPRICH; HÜBNER, 2017; STORNAIUOLO; SANTAMBROGIO; SCIUTO, 2018; GU; HAO; YANG, 2018) utilizaram FPGAs com processadores ARM por serem capazes de inicializar totalmente sistemas Linux sem uma prévia programação lógica da FPGA. Isso permite que os desenvolvedores executem softwares e suas aplicações, em uma FPGA, em linguagem de programação C ou, mais recentemente, Python, sem a necessidade de usar a linguagem HDL que é destinada a projetos de circuitos lógicos digitais.

Além disso, atualmente existem ferramentas e frameworks mais confiáveis para fornecer suporte para síntese de alto nível (HLS), auxiliando o desenvolvimento de aceleradores em linguagens de alto nível (CONG et al., 2011; CANIS et al., 2011). Como resultado, esses dispositivos estão atraindo a atenção de desenvolvedores de softwares não tradicionais no campo das FPGAs. Esses conceitos e princípios de design possibilitam abstrações de alto nível em um fluxo de desenvolvimento FPGA, facilitando seu uso (SKALICKY et al., 2018) e reduzindo o custo de desenvolvimento de hardware, que pode demandar um ciclo de desenvolvimento mais demorado e mais oneroso, se comparado a projetos de desenvolvimento somente em softwares (JANSSEN; ZIMPRICH; HÜBNER, 2017; HONDA; WEI; AMANO, 2019).

Em 2017, a Xilinx lançou um projeto de código aberto chamado PYNQ (CORRADI, 2018). Considerando a linguagem Python como uma linguagem de grande popularidade e de interesse para desenvolvedores de softwares (CASS, 2018), o projeto PYNQ permite maior produtividade de aplicações Python, com suporte ao Jupyter Notebooks, em uma plataforma de classe System on Chip (SoC), programável, que integra um processador multicore (Dual-core ARMő Cortexő-A9) e um FPGA da família *Zynq* em um único circuito integrado. Essa arquitetura heterogênea de hardware e software do Zynq SoC permite que os desenvolvedores de softwares explorem recursos de hardware diretamente na linguagem Python. Segundo Junchao et al. (JUNCHAO; WEILIANG; SHAOJUN, 2001), SoCs são usados para reduzir o tempo de ciclo do produto e o custo de desenvolvimento. Nesse sentido, os núcleos Intellectual Property (Intellectual Property (IP)) "são projetos digitais contendo hardware dedicado, pré-projetado, pronto para ser utilizado em SoCs programáveis contendo FPGAs.

Portanto, os núcleos IP são os blocos lógicos reprogramáveis criados com Vivado HLS (C++) e a customização dos blocos lógicos gerados podem ser reutilizada para implementação em hardware, principalmente nos projetos baseados em FPGA, os quais podem fornecer prontamente suas funções otimizadas para uso com o mesmo nível de abstração das bibliotecas de software. Essas bibliotecas, que representam a customização do núcleo IP, permitem a exploração da arquitetura heterogênea de hardware do Zynq SoC e são chamadas de *Overlays*, ou bibliotecas de hardware. A *Overlays* suporta as interfaces de entrada e saída da placa, controladores de memória e pode ser estendida para hardware de núcleo IP, personalizado adicional. Desenvolvedores de software podem desenhar projetos dessa natureza através de ferramentas e frameworks utilizando a linguagem de programação C ou, mais recentemente, Python, sem a necessidade de utilizar a linguagem HDL, que é destinada a projetos de circuitos lógicos digitais.

Além disso, já existem ferramentas e frameworks mais confiáveis para fornecer suporte para síntese de alto nível (HLS), auxiliando no desenvolvimento de aceleradores em linguagens de alto nível (CONG et al., 2011; CANIS et al., 2011). Como resultado, esses dispositivos atraem a atenção de desenvolvedores de software não tradicionais na área

de FPGAs. Esses conceitos e princípios de design permitem abstrações de alto nível em um fluxo de desenvolvimento de FPGA, facilitando seu uso (SKALICKY et al., 2018) e reduzindo o custo de desenvolvimento de hardwares. Outros métodos, como os projetos desenvolvidos em HDL, podem exigir um ciclo de desenvolvimento mais longo, conhecimento mais específico e, portanto, podem ser mais caros em comparação, apenas, com os projetos de desenvolvimento de software (JANSSEN; ZIMPRICH; HÜBNER, 2017; HONDA; WEI; AMANO, 2019).

A PYNQ-Z1 possui barramentos internos compartilhados de alta velocidade para comunicação e acesso entre os microprocessadores e o FPGA, com memórias e periféricos externos compartilhados, presentes no dispositivo. Essa interface de comunicação é denominada de Advanced Microcontroller Bus Architecture (*Advanced Microcontroller Bus Architecture* (AMBA)), e é capaz de fornecer alto desempenho e alta banda de dados com baixa latência por meio do protocolo Advanced eXtensible Interface (AXI). Tanto a interface AMBA quanto o protocolo *Advanced eXtensible Interface* (AXI) foram desenvolvidos pelo fabricante ARM. O protocolo é baseado em transferências em rajadas (*burst*), com cinco canais independentes para a transferência de endereço de escrita, endereço de leitura, dados de escrita, dados de leitura e resposta de escrita, permitindo leituras e escritas simultâneas, bem como disparar múltiplas requisições de endereços de escritas e leituras. A interface é amplamente utilizada na interconexão de periféricos com um ou mais processadores em dispositivos e segue o padrão mestre-escravo, podendo o barramento conter múltiplos mestres e múltiplos escravos, possuindo também um interconector para mediar as transações entre o sistema de processamento e seus periféricos.

Para a quarta versão da especificação, AMBA AXI4, a interface implementa três variações do protocolo:

- AXI4-*Full*, ou apenas AXI4, contemplando a especificação completa do protocolo, com seus acessos mapeados em memória, possibilitando transações de *burst*;
- AXI4-*Lite*, um subconjunto mais simples da especificação, com seus acessos mapeados em memória e simples implementação em hardware, porém sem suporte a transações de *burst*;
- AXI4-*Stream*, protocolo para transferência de dados em cadeia, unilateral, entre periféricos.

2.5 Trabalhos Correlatos

A abordagem proposta de hibridização e paralelização de um algoritmo de agrupamento de dados foi o cerne da pesquisa por meio de uma ferramenta de alto nível, a qual é o estado da arte para implementação de hardware e permite a qualquer usuário implementar hardware dedicado. Portanto, para a implementação, a PYNQ-Z1 foi utilizada

como plataforma principal do projeto. Uma boa fonte de documentação é fornecida pelo fabricante ("PYNQ", 2022), possibilitando entender melhor toda a tecnologia envolvida nesse contexto. Muitos experimentos são publicados em comunidades de usuários como Git ou similares também.

Nessa direção, esta pesquisa buscou investigar a simplicidade e a facilidade de hibridização do algoritmo BH por não haver parâmetros, e apresentar o melhor desempenho de uma implementação paralelizável do algoritmo BH em hardware. Além disso, foi planejado chegar a uma implementação por meio de uma ferramenta de alto nível que permitisse a qualquer usuário sem muita experiência em hardware dedicado replicar tal abordagem. Assim, a complexidade necessária para a criação de um hardware dedicado para aceleração das tarefas de agrupamento de dados foi reduzida de forma considerável, pois utilizamos técnicas de paralelismo em hardware, utilizando o HLS.

Tal fato é importante, pois uma das contribuições deste trabalho é uma abordagem simples para aproveitar projetos bons, igualmente simples e, dessa forma, abstrair dificuldades em criar aceleradores em hardware com técnicas de paralelismo, podendo inspirar pesquisadores a implementarem outras variantes como conjuntos de dados de dimensões maiores e novos algoritmos, atendendo a uma plataforma de alto nível, que é o estado da arte no campo das aplicações SoCs, proporcionando tempo viável de execução com uma boa acurácia.

O algoritmo BH original foi proposto para o problema de agrupamento de dados. Muitas variantes do BH foram introduzidas no grupo como exemplo, o algoritmo BH foi usado para acelerar a velocidade de agrupamento de dados, por software e hardware (TSAI; HSIEH; CHIANG, 2015). Um dos principais motivos pelos quais os métodos de agrupamento tradicionais são ineficientes para analisar conjuntos de dados em larga escala reside no fato de a maioria deles ser projetado para um sistema centralizado. Isso significa que se o tamanho dos dados de entrada exceder o tamanho do armazenamento ou memória de tal sistema, a tarefa de agrupar se torna muito mais difícil.

Para mitigar esse problema, Tsai et al. (2015) propuseram um eficiente algoritmo de agrupamento de dados, o *MapReduce Black Hole* (MRBH), tendo em vista alavancar a força do algoritmo BH, usando um modelo de programação *MapReduce* do *Hadoop* para acelerar a velocidade de agrupamento por software e hardware. Usando o *MapReduce*, o MRBH dividiu um grande conjunto de dados em vários conjuntos de dados pequenos e definiu o agrupamento desses dados menores aplicando processamento paralelo. Além disso, a MRBH herdou as características do BH, o que significa que nenhum parâmetro precisou ser ajustado manualmente e, portanto, a implementação tornou-se mais fácil. Para avaliar o desempenho do algoritmo proposto, vários conjuntos de dados foram usados com diferentes números de grupos. Resultados experimentais mostraram que o algoritmo proposto forneceu uma aceleração significativa conforme o número de grupos aumentou.

Outros autores, como Nemati et al. (NEMATI; MOMENI; BAZRKAR, 2013), estu-

daram adaptações no BH para avaliar seu desempenho e resolver funções de *benchmarks* conhecidas, constatando resultados promissores diante de algoritmos similares como o algoritmo genético. Eskandarzadehalamdary et al. (ESKANDARZADEHALAMDARY; MASOUMI; SOJODISHIJANI, 2014) relataram uma combinação entre algoritmo K-médias e BH para resolver um problema de agrupamento de dados e os resultados obtidos, bem como o desempenho da solução foram superiores, quando comparados aos algoritmos tradicionais K-médias, BH e PSO (ESKANDARZADEHALAMDARY; MASOUMI; SOJODISHIJANI, 2014). Pashaei e Aydin (PASHAEI; AYDIN, 2017), por sua vez, propuseram o algoritmo BH adaptado para resolver um problema de classificação e seleção de características de dados biológicos. Seus resultados tiveram maior performance computacional e atingiram-se resultados melhores em relação ao algoritmo PSO adaptado e o algoritmo genético.

Kumar et al. pesquisou sobre o algoritmo BH e suas aplicações (KUMAR; DATTA; SINGH, 2015). Nesse estudo, discutiu-se sobre a natureza metaheurística, avaliando a importância de balancear a busca da solução, de modo diversificada e intensa. Para uma busca diversificada são utilizados métodos estocásticos; essa forma exploratória aumenta as chances de o algoritmo não convergir em um ponto mínimo local.

Já a busca intensificada utiliza informações conhecidas da população para se aproximar de um ponto ótimo global. O autor cita as vantagens do algoritmo BH em relação a outros algoritmos semelhantes, pois este não apresenta problemas de ajuste de parâmetros como o algoritmo genético, que precisa de ajustes durante as operações genéticas. No estudo foi também tratada a questão da eficiência do algoritmo em convergir para um ponto ótimo global e sua simplicidade de implementação, posto ser a estrutura do algoritmo simples.

A pesquisa discutiu, ainda, o problema que é objeto deste estudo, um problema de agrupamento de dados, relatando que, embora o algoritmo K-médias seja simples e eficiente, não está livre de convergir em um ponto mínimo local.

Ibrahim et al. (IBRAHIM et al., 2018) também efetuaram uma pesquisa sobre as aplicações do algoritmo BH em diversas áreas como a inteligência artificial, projetos de engenharia, engenharia de software, indústria, engenharia civil, pesquisa operacional, processamento de imagens, rede de sensores sem fio, programação de tarefas, otimização de parâmetros, sistema de energia, agrupamento de dados, análise médica etc. Devido à popularidade do algoritmo BH, o algoritmo pode ser explorado em novas áreas tais como projetos envolvendo sua implementação em hardware.

Para uma comparação mais justa, a pesquisa buscou demonstrar os tempos de execução e acurácia de alguns algoritmos e, desse modo, propomos uma nova abordagem, a qual obteve redução significativa no tempo de execução, sem que houvesse redução da qualidade relacionada ao agrupamento de dados. Conforme alguns autores já mencionados (HATAMLOU; ABDULLAH; HATAMLOU, 2011; HATAMLOU, 2013; NIKBAKHT; MIRVAZIRI, 2015), os mesmos algoritmos foram utilizados neste estudo; no entanto, não

há informações sobre seus tempos de execução. Além disso, os resultados deste estudo contribuem significativamente para trabalhos futuros, pois fornecem todos os parâmetros utilizados. Parâmetros como o tamanho da população e o número de iterações para atingir um valor ótimo global não foram relatados em estudos semelhantes (HATAMLOU; ABDULLAH; HATAMLOU, 2011; HATAMLOU, 2013; NIKBAKHT; MIRVAZIRI, 2015; HATAMLOU; ABDULLAH; NEZAMABADI-POUR, 2011; ESKANDARZADEHALAMDARY; MASOUMI; SOJODISHIJANI, 2014; BIJARI et al., 2018; HAN et al., 2017; DOWLATSHAHI; NEZAMABADI-POUR, 2014; SAHOO et al., 2014). Esses parâmetros são fundamentais para entender completamente os resultados considerados ótimos. Dessa maneira, justificou-se a replicação de alguns algoritmos para tal validação temporal, na qual o algoritmo BH obteve um desempenho melhor que os demais nos testes de software, optando-se, então, por implementá-lo em hardware.

Hatamlou et al. (2011) propuseram um método de otimização conhecido como algoritmo *Big Bang - Big Crunch*, para resolver o agrupamento de dados de quatro *benchmarks* conhecidos, *Iris*, *Wine*, *Contraceptive Method Choice* (CMC) e *Wisconsin Breast Cancer* (BLAKE; MERZ, 1998). Os resultados foram comparados aos algoritmos PSO, Algoritmo Genético (AG) e Kmédias, mostrando que o algoritmo *Big Bang - Big Crunch* obteve resultados de qualidade superior em relação às métricas de agrupamento adotados.

Utilizou-se a soma da distância intragrupos, tomando nota do melhor valor obtido, ou seja, o menor valor da soma da distância intragrupos, o valor médio da soma conforme a repetição dos testes, bem como o pior valor da soma e desvio padrão da solução. O algoritmo BB-BC mostrou ser uma técnica adequada e confiável para agrupamento de dados, pois tem uma estrutura simples e fornece grupos de alta qualidade em termos de soma das distâncias intragrupos. Posteriormente, Hatamlou propôs mais um estudo comparativo entre os algoritmos K-médias, PSO e BB-BC e incluiu em seus estudos os algoritmos GSA e BH (HATAMLOU, 2013), todos esses para resolver o problema de agrupamento de dados usando seis conjuntos de dados conhecidos, adicionando, em relação ao estudo supracitado, as bases do problema *Glass* e *Vowel*, sendo seus resultados comparados.

O algoritmo BH teve qualidade superior nas soluções, ante os outros algoritmos e em relação às mesmas métricas de agrupamento. No estudo, o algoritmo BH teve destaque por ser livre de parâmetros, de fácil entendimento e apresentando uma estrutura simples de ser implementada. O único contraponto, mesmo que por uma diferença discreta, o BB-BC foi superior aos demais para resolver a base de dados *Wisconsin Breast Cancer*. Seus resultados foram comparados utilizando métodos estatísticos, como o teste de *Friedman*, *Iman-Daven Port* e pós-teste de *Holm*, os quais permitiram concluir a superioridade do algoritmo BH sobre os demais quanto a resolver os conjuntos de dados envolvidos no estudo.

Capítulo 3

Metodologia e Descrição da Implementação

3.1 Desenvolvimento do Acelerador em Hardware

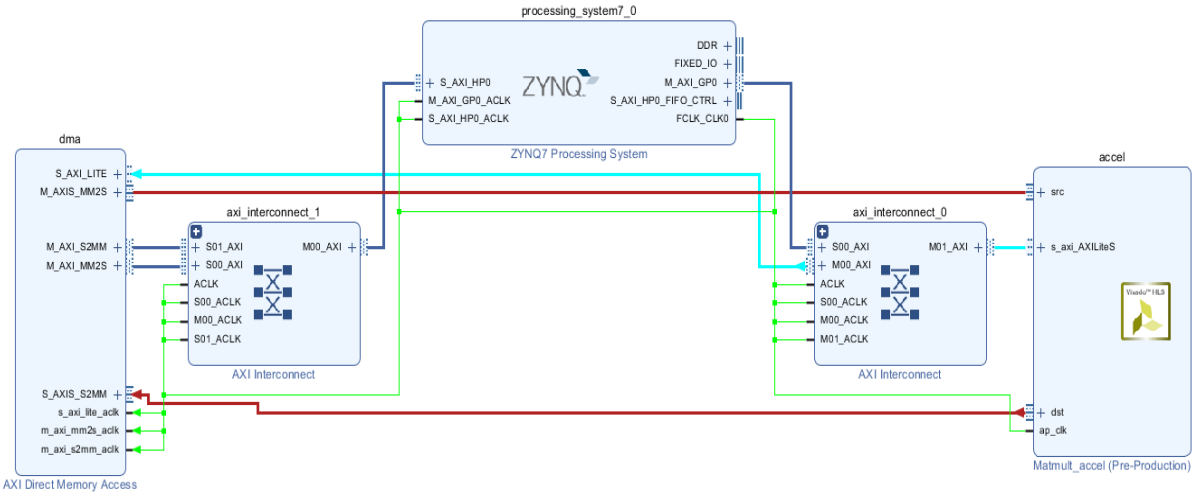
No desenvolvimento do projeto do acelerador em hardware, foram gerados quatro núcleos IP, um para cada conjunto de dados, utilizando a ferramenta Vivado HLS versão 19.2, em linguagem de programação C++, para sintetizar o projeto. O projeto Vivado consistiu em um projeto de programação lógica (programação lógica (PL)) criado em Vivado HLS e a definição correspondente das configurações do sistema de processamento (sistema de processamento (SP)). Isso ocorre porque o bloco de design padrão inclui *Overlays* de interfaces de entrada e saída e controladores de acesso direto à memória (DMA), já fornecidos na configuração inicial da plataforma PYNQ. Alguns trabalhos realizados demonstram implementações e aplicações de projetos desta natureza (CONG et al., 2011; CANIS et al., 2011). O código desenvolvido e utilizado para implementar os quatro núcleos IP está no Apêndice A.

Na implementação foi desenvolvido um bloco design que forneceu uma estrutura de controle de periféricos ou *Memory-Mapped Input/Output* (*Memory-Mapped Input / Output* (MMIO)), utilizada para comunicação entre SP e PL quando o desempenho computacional não é crítico. Para transferência de grandes quantidades de dados entre o SP e PL, foram utilizadas interfaces Zynq de alta performance com IP DMA e a classe Pynq DMA. Assim, o projeto Vivado foi criado e um novo núcleo IP com lógica customizada pôde ser desenvolvido, sintetizado e implementado, e dois arquivos foram gerados: um arquivo *bitstream*, que descreve os dados de configuração a serem carregados no Zynq SoC e um arquivo Tool Command Language (Tcl) do projeto do bloco lógico final, que contém

um *script* Tcl, consistindo em funções Tcl e usadas como comandos. *Tool Command Language* (Tcl) é uma linguagem de programação interpretada com variáveis, procedimentos (*procs*) e estruturas de controle para fazer interface com uma variedade de ferramentas de design e para os dados de projeto (XILINX, 2014).

A Figura 1 mostra o núcleo IP com interface AXI gerado no Vivado Design Suite, versão 19.2. Foi utilizada a classe Python MMIO para simplificar a comunicação entre Zynq e o processador ARM9, representados pelas linhas na cor azul escuro. Uma vez que o núcleo IP foi criado e o mapa de memória reconhecido através do Vivado Design Suite, o módulo MMIO permitiu o acesso aos locais mapeados na memória na programação lógica (PL). O fluxo dos dados matriciais deste estudo ocorre pelas conexões AXI4-*Stream* e está destacado na cor vermelha. As linhas verdes representam a comunicação dos sinais de *clock* e as as linhas na cor azul claro representam as conexões AXI4-*Lite*.

Figura 1 – Bloco Design do núcleo IP desenvolvido para o projeto.



Fonte: Vivado Design Suite v2019.2

Existem outras plataformas SoC com capacidade superior ao Pynq-Z1 que estão disponíveis comercialmente. No entanto, o Pynq-Z1 apresenta melhor custo-benefício, pois incorpora tecnologia moderna e permite implementação de alto nível. Porém, algumas limitações devem ser mencionadas, pois influenciaram na definição do fluxo de dados e suas respectivas representações. Na solução proposta, a mesma memória do dispositivo, com capacidade de 512 MB, é compartilhada com o sistema operacional Linux.

Devido a essa limitação, os conjuntos de dados e as soluções candidatas não foram representados em uma única matriz dedicada. Alguns testes preliminares no Pynq foram realizados, evidenciando limitações de memória para matrizes maiores que 120 x 120. Tais testes foram baseados em um projeto de um acelerador em hardware para multiplicação de matrizes (BAGNI et al., 2016). Assim, foi definido que os conjuntos de dados e as

soluções candidatas seriam bem particionados em várias submatrizes, otimizando o uso da memória disponível e compensando os custos de acesso a memória através do acelerador em hardware. O particionamento das matrizes será detalhado na seção 3.2.

3.2 Estrutura de Dados para Agrupamento dos Dados

Inicialmente, realizou-se uma investigação sobre os conjuntos de dados utilizados nesse estudo e a qualidade das soluções obtidas com os algoritmos BH, PSO, BB-BC e GSA para agrupamento de dados avaliados. A partir desse estudo preliminar, foi adquirido conhecimento para avaliar os resultados do estudo. Para cada problema, foi adquirido um agrupamento; diferentes dimensões de problema foram explorados, conforme mostrado na Tabela 1. Os conjuntos de dados usados neste estudo são encontrados no *UCI Machine Learning Repository* (DUA; GRAFF, 2017).

Tabela 1 – Principais características das bases de dados.

Base	Nž de grupos	Nž de dimensões	Nž de objetos
Iris	3	4	150 (50,50,50)
Wine	3	13	178 (59,71,48)
Glass	6	9	214 (70,76,17,13,9,29)
CMC	3	9	1473 (629,334,510)

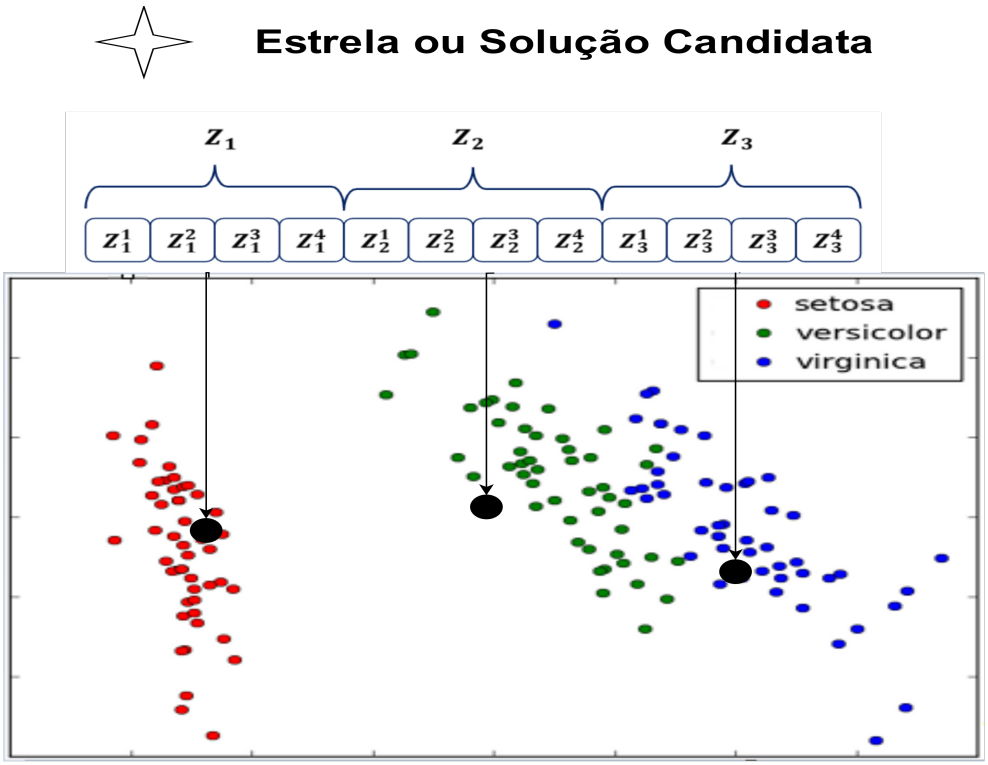
Os algoritmos estudados tomam como ponto de partida a geração de uma população inicial de candidatos à solução ótima, espalhados em localizações aleatórias no espaço de busca. O tamanho da solução candidata depende dos dados de cada problema. Para tanto, a estrutura da solução candidata deve ser ajustada de acordo com o número de grupos e características de cada conjunto de dados. Cada solução candidata é considerada como tendo k centroides iniciais dos grupos e a unidade individual da matriz como a dimensão central do grupo. A Figura 2 ilustra uma solução candidata para o conjunto de dados *Iris*, que tem três grupos e 150 objetos de $4 - dimenses$.

O número de recursos de um centroide define o comprimento de um grupo; o número de grupos define o comprimento da matriz unidimensional e o valor pode ser alterado, dependendo do tamanho do problema a ser otimizado.

Essa estrutura matricial ofereceu, ao projeto, flexibilidade para suportar qualquer formato numérico, considerando o exemplo do conjunto de dados *Iris*, onde Z_1, Z_2 e Z_3 são os centroides. Essa flexibilidade também foi importante, uma vez que a função de aceleração foi implementada para processar duas matrizes previamente estruturadas: conjunto de dados e população.

Conforme mencionado, de acordo com os testes preliminares, o presente estudo propôs trabalhar com matrizes de tamanho fixo (120x120), a fim de não haver problemas com as

Figura 2 – Solução candidata para o conjunto de dados *Iris*. Neste caso, o espaço de busca é de 4 dimensões.



Fonte: Elaboração do autor (2022)

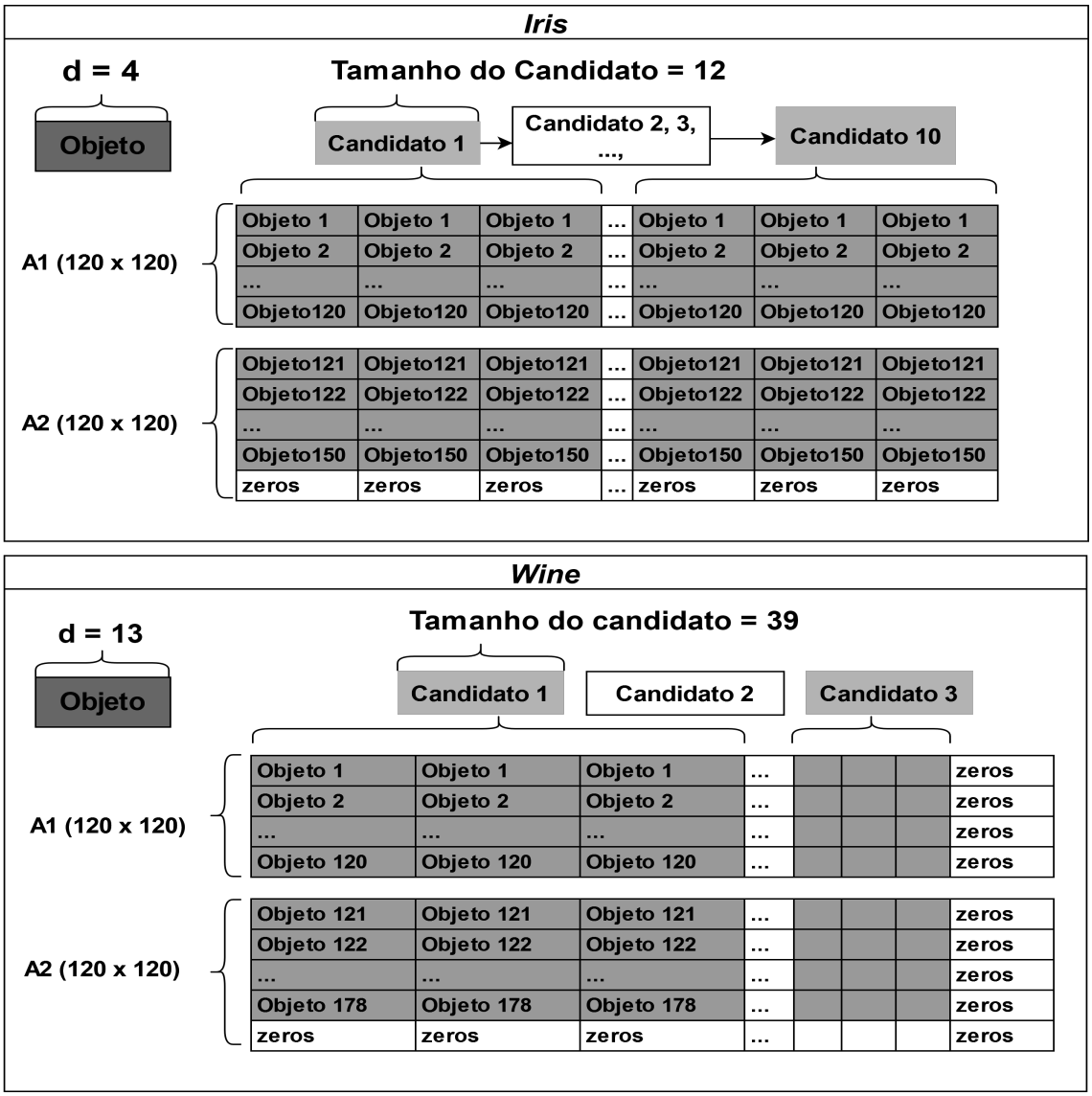
limitações de recursos de memória do Zynq SoC. A representação dos dados das submatrizes dos conjuntos *Iris* e *Wine* são mostradas na Figura 3.

Portanto, as matrizes que representam os conjuntos de dados e a população de soluções candidatas foram ajustadas em matrizes 120 x 120 e estruturadas para usar operações matriciais de forma otimizada.

O conjunto de dados *Iris* poderia ser colocado, por exemplo, em uma matriz de 150 linhas, representando os 150 objetos, e quatro colunas representando a dimensão de cada objeto. No entanto, a fim de contornar a questão de limitação de memória e ainda para explorar o paralelismo computacional baseado na operação de subtração, onde cada elemento da matriz resultante pode ser calculado de forma simultânea , os conjuntos de dados *Iris*, *Wine* e *Glass* foram representados por duas submatrizes $A1$ e $A2$. Para o conjunto de dados *CMC*, que tem mais de 240 objetos, 13 submatrizes foram necessárias para representar seus 1473 objetos. A codificação em Python para montagem das submatrizes dos conjuntos de dados é mostrada no Apêndice B.

Conforme características dos conjuntos *Iris* e *Wine*, utilizadas como exemplo na Figura 3, a solução candidata para os dois casos é representada por três grupos e cada grupo tem o mesmo tamanho do objeto do respectivo conjunto de dados. Para explicar de forma visual e intuitiva o funcionamento das operações matriciais, os objetos foram destacados

Figura 3 – Estrutura da matriz ajustada para conjuntos de dados *Iris* e *Wine*.



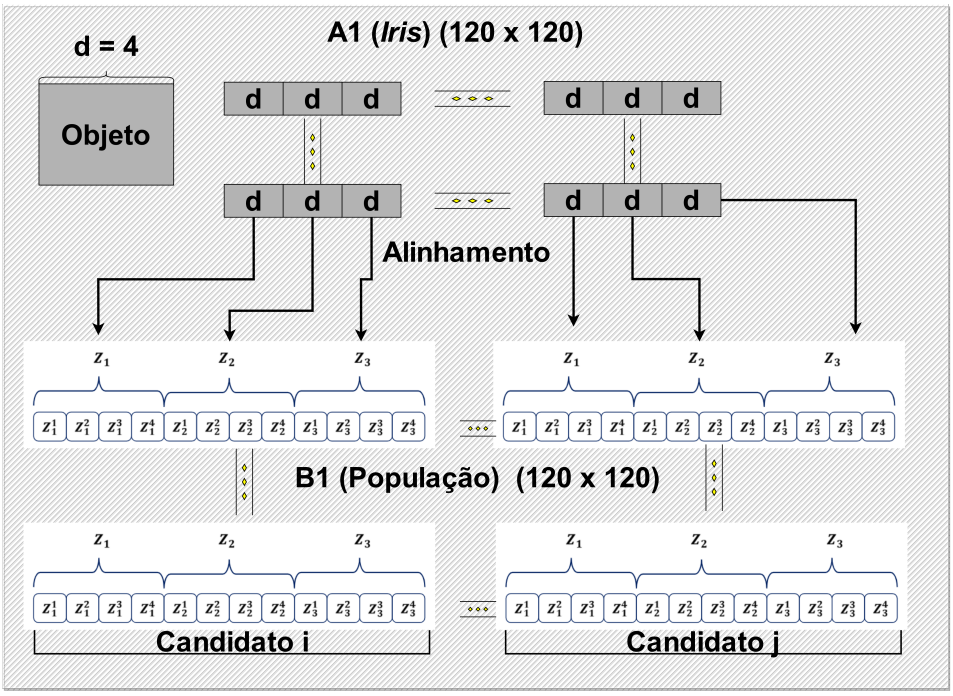
Fonte: Elaborado pelo autor (2022)

em blocos, apresentando como os dados foram disponibilizados nas submatrizes. Dessa forma, a estrutura das submatrizes foram alinhadas conforme mostra a Figura 4, na qual foi considerado o caso do conjunto *Iris* para mostrar o alinhamento.

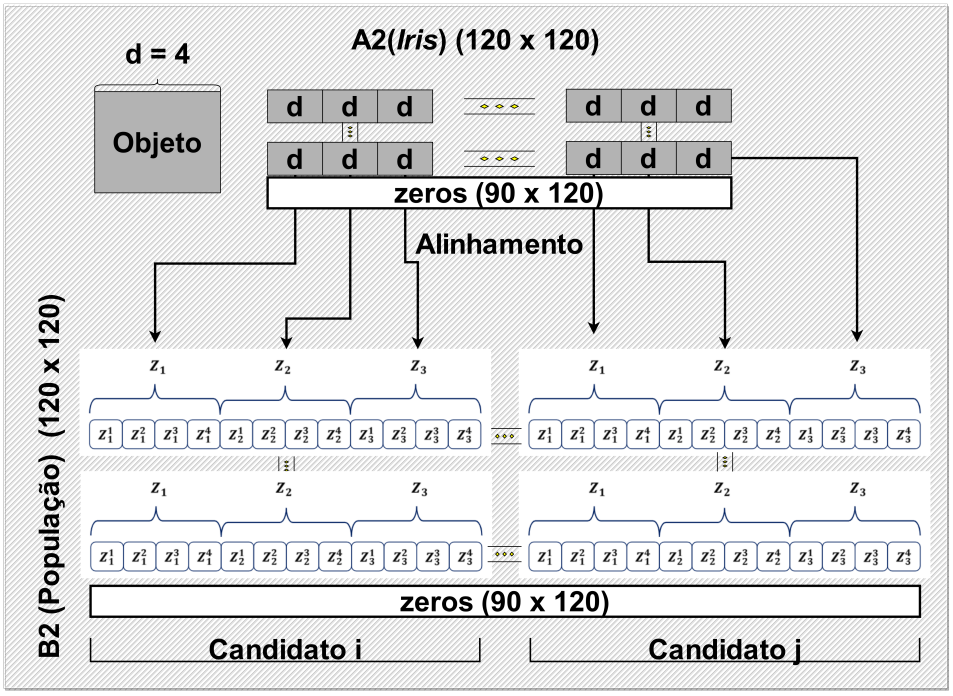
Os centroides de uma solução candidata foram destacados como Z_1, Z_2 e Z_3. As 120 linhas da submatriz $B1$ da população são preenchidas com a mesma solução candidata. A próxima solução candidata é concatenada nas mesmas 120 linhas, preenchendo as próximas colunas vagas da submatriz $B1$ e assim sucessivamente, até o limite de 120 colunas.

Desse modo, a submatriz $A1$ do conjunto *Iris* representa os primeiros 120 objetos para 10 soluções candidatas, as quais podem ser representadas nas submatrizes $B1$, uma vez que cada solução candidata tem um tamanho de 12. O alinhamento da submatriz $A1$ e da

Figura 4 – Estrutura de uma submatriz populacional e alinhamento com a submatriz do conjunto de dados.




```
Calcula as distâncias entre objetos
e
os centroides das soluções candidatas (i,i+1,i+2,...,j)
```



Fonte: Elaborado pelo autor (2022)

submatriz $B1$ permitiu a otimização da operação matricial que armazenou, em um vetor de saída, os resultados da função objetiva das 10 primeiras soluções candidatas para os 120 primeiros objetos do conjunto *Iris*.

A representação da submatriz $A2$, que serve para dar continuidade na representação do conjunto de dados, mostra o caso em que lacunas vazias foram preenchidas com o valor zero para anular quaisquer somas indesejadas nas operações matriciais. Essas lacunas vazias são mantidas para atender a suposição de tamanho de matriz fixa. Na submatriz $B2$ estão representados os mesmos candidatos à solução da submatriz $B1$, servindo para completar o cálculo da função objetiva, considerando-se os objetos do conjunto de dados que ficaram fora da submatriz $A1$ e estão na submatriz $A2$. Porém, seguindo as mesmas premissas da submatriz $A2$, a submatriz $B2$ também possui lacunas vazias. Portanto, o número de submatrizes necessárias para representar um determinado tamanho da população depende da relação entre o tamanho da solução candidata e o tamanho fixo das submatrizes.

A Figura 5 mostra a relação entre as características de cada conjunto de dados com a definição da montagem das submatrizes. O tamanho de cada solução candidata refere-se a K grupos multiplicado pelo tamanho d do objeto. Assim, dimensiona-se quantas soluções candidatas podem ser representadas por submatriz e, conforme as premissas da montagem das submatrizes e definição prévia do tamanho da população, calcula-se a quantidade X de submatrizes para representar toda a população. A Tabela 2 mostra a relação entre o número e o tamanho das soluções candidatas por submatriz, bem como o número de submatrizes necessárias para representar o tamanho da população.

De acordo com a tabela acima, como se pode notar, X submatrizes B são geradas para representar toda a população candidata e Y submatrizes A são geradas para representar os objetos. Nesse contexto, uma estrutura de repetição para sequenciar as entradas do acelerador no hardware foi implementada para calcular o valor objetivo de todos os candidatos à solução, na qual um laço de repetição controla a sequência das submatrizes populacionais e outro laço de repetição controla a sequência das submatrizes do conjunto de dados. Esses laços de repetição se fizeram necessários, uma vez que o acelerador de hardware carrega duas matrizes A e B, por vezes, um buffer de entrada. Para cada sequência de entradas no buffer, são retornados valores parciais das soluções candidatas. Ao final de todas as submatrizes X e Y, e de acordo com o controle sequencial, os resultados são disponibilizados em uma matriz unidimensional com todos os valores objetivos da população.

Esse processo de montagem das submatrizes da população é utilizado no início do algoritmo BH, como será detalhado melhor na seção 3.3. Entretanto, a cada iteração do algoritmo BH, há situações em que a localização das soluções candidatas deve ser atualizada. Uma delas é quando todas as soluções candidatas da população se movimentam no espaço de busca, mimetizando a força gravitacional do buraco negro, atraindo todas as estrelas do espaço (Eq. 20).

Tabela 2 – Relação dos parâmetros usados neste trabalho (Número de soluções candidatas por submatriz e o tamanho da candidata e número de submatrizes por tamanho da população).

Base	Candidatos / Tamanho do Cand.	População	Submatrizes
Iris	10 / 12	200	20
Wine	3 / 39	201	67
Glass	2 / 54	200	100
CMC	4 / 27	200	50

Fonte: Elaborado pelo autor (2022)

Outra situação é quando uma estrela é sugada pelo buraco negro, entrando a estrela na região do horizonte de eventos (19) do buraco negro. Tal estrela ou solução candidata é substituída por uma nova estrela, gerada aleatoriamente no espaço de busca. Logo, para manter as premissas de representação dos dados por submatrizes, uma função foi implementada com o objetivo de atualizar as submatrizes sempre que ocorrer tais situações. O código desenvolvido em Python para definição das funções que geram um candidato à solução, assim como a montagem da população e atualização das submatrizes, está mostrado no Apêndice C. Sempre que um candidato é deslocado no espaço de busca ou substituído por um novo, a representação das submatrizes é atualizada.

A Figura 6 mostra o mecanismo usado para otimizar o cálculo da distância entre cada solução candidata e os objetos do conjunto de dados. As operações matriciais otimizaram a forma de calcular as distâncias euclidianas dos objetos aos centroides de cada candidato; cada linha da submatriz A é subtraída elemento a elemento da linha da submatriz B e cada elemento resultante é elevado ao quadrado.

No final, o valor é agregado por meio de uma soma e a raiz quadrada é extraída. A agregação é controlada pelo tamanho d do objeto e k grupos. A matriz resultante permitiu que o centróide mais próximo de cada objeto fosse conhecido, sendo representado na etapa "Selecione a menor distância, e uma matriz unidimensional chamada "*pool*" foi obtida com o valores da função objetivo de todos os candidatos representados na matriz populacional.

3.3 Implementação do Algoritmo *Black Hole*

O algoritmo BH foi implementado em Python, versão 3.6 (VANROSSUM; DRAKE, 2010) usando-se um interpretador interativo denominado Jupyter Notebooks ("JUPYTER", 2022), ambos fornecidos pela Pynq-Z1 ("PYNQ", 2022). A Figura 7 ilustra o diagrama geral da solução proposta. Os elementos destacados com fundo preto são representações dos dispositivos físicos: um multiprocessador ARM9, uma memória DRAM de 512 MB e blocos lógicos gerados na *Overlay*. Os blocos lógicos estão agrupados dentro de um retângulo e representa a estrutura do acelerador em hardware para o cálculo da função objetiva da população de estrelas. As setas representam o fluxo de dados, demonstrando,

de forma abstrata, a comunicação dos blocos lógicos. O sistema de processamento Zynq, da família 7000, comunica-se com o acelerador e a estrutura de acesso direto à memória (DMA) através de blocos de interconexões com barramento AXI. O fluxo de dados ocorre pela estrutura de barramentos AXI e interfaces de alta performance, nas quais tem influência direta quanto ao ganho de performance da solução, uma vez que os dados matriciais são transferidos em cadeias para o acelerador pelo protocolo AXI4-*Stream*.

O mapeamento de conexões endereçadas na memória, os sinais de controle e configurações mais simples utilizam outros barramentos com os protocolos AXI-4 e AXI4-*Lite*. As etapas iniciais do algoritmo BH são a montagem das submatrizes do conjunto de dados a ser agrupado e a população de estrelas, e podem ser observadas no topo do diagrama geral.

Todos os dados matriciais são armazenados na memória DRAM e são colocados como dados de entrada na *Overlay* através do bloco DMA.

Na parte inferior do diagrama geral é ilustrado um vetor de saída da *Overlay* denominado "*pool*". Nesse vetor de saída são armazenados os resultados da função objetiva de cada estrela da população. Logo, o vetor *pool* possui o mesmo comprimento do número de estrelas da população. O valor da função objetiva da primeira estrela é armazenado na primeira coluna do vetor. O valor da próxima estrela é armazenado na próxima coluna e, assim, sucessivamente, até armazenar o valor da função objetiva da última estrela da população.

Com isso, o algoritmo volta a ser processado no nível de software, no qual o menor valor obtido e a localização da respectiva estrela são selecionados pelo BH. Calculam-se as novas localizações das estrelas da população, conforme a equação 20, que simula o buraco negro atraindo, com sua força gravitacional, as demais estrelas no espaço de busca.

Desse modo, conforme o tamanho do raio de ação calculado pela equação 19, é verificado se alguma estrela se encontra a uma distância menor do que o raio de ação do buraco negro. Se sim, uma nova estrela é gerada em um local aleatório no espaço de busca, substituindo a estrela que foi sugada pelo buraco negro. Então, inicia-se uma nova iteração do algoritmo BH com a população de estrelas resultante da última iteração. Quando o número máximo de iterações é atingido, os melhores centroides obtidos são usados para o agrupamento de dados. O código do algoritmo BH implementado na linguagem Python se encontra no Apêndice C.

As principais etapas do algoritmo BH são apresentadas na Figura 8. A etapa que calcula a função objetivo (f_i) das soluções candidatas representa a implementação híbrida em hardware e é utilizada por meio das *Overlays* geradas para este estudo. O acelerador de hardware, ou *Overlay*, é chamado pela função "*Accel*". O código *Overlay* implementado na linguagem Python se encontra no Apêndice D desse estudo. As etapas restantes são processadas em nível de software. Quando o número máximo de iterações é atingido, os melhores centroides obtidos são usados para o agrupamento de dados.

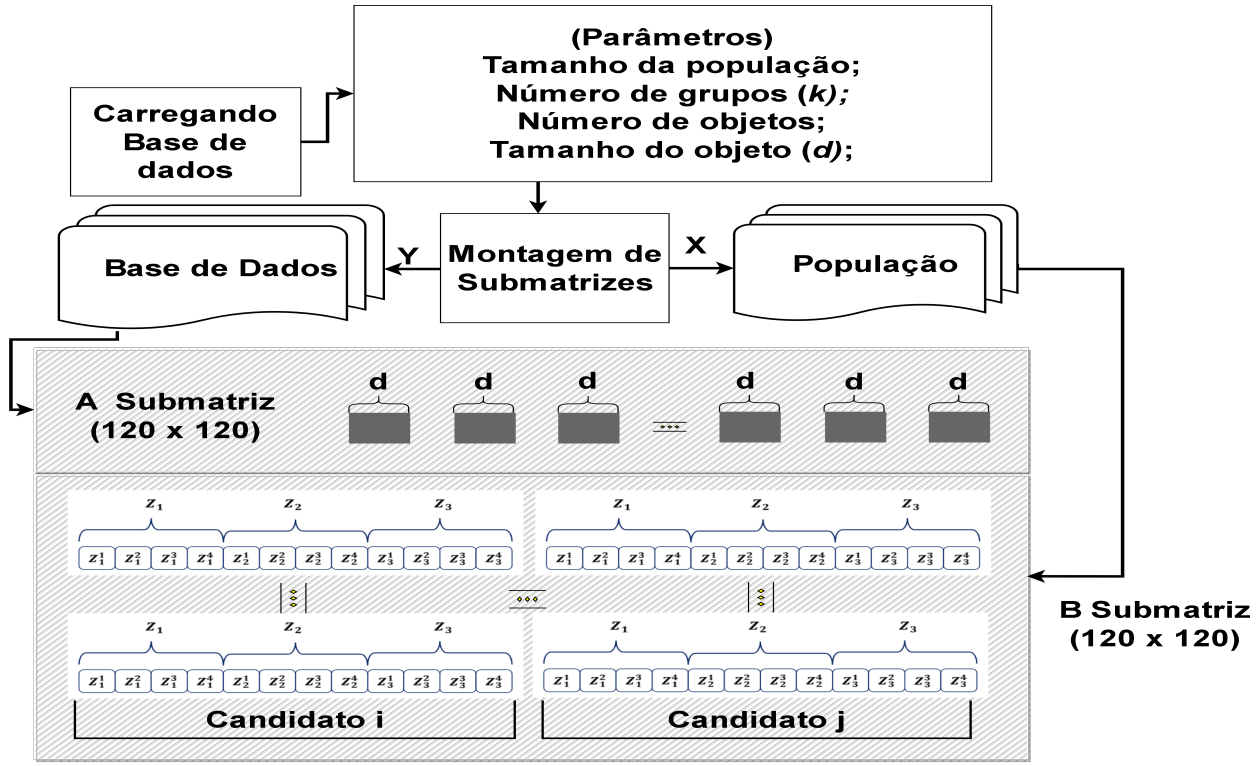
Conforme mencionado, há uma relação entre as características de cada conjunto de dados com a definição da montagem das submatrizes. Essa função personalizada é denominada "*Assembly*" e é usada pela etapa "*Gera população de estrelas*" . Os passos "*Calcule a nova posição da estrela, Equação 20*" e "*substitua X_i por uma nova estrela em um local aleatório no espaço de busca*"seguem as premissas da montagem das submatrizes para atualizar os dados e essa função customizada se chama "*update*"(Ver o uso das funções *Assembly* e *update* na Figura 8).

A abordagem atual não se limita a trabalhar apenas com os conjuntos de dados fornecidos, mas também permite trabalhar com conjuntos de dados mais complexos. Respeitando as limitações mínimas na representação de um candidato para a solução de tamanho $k*d$, o tamanho da solução não pode ser maior que 120 para a implementação atual. Qualquer produto da multiplicação entre d e k, com resultado menor que 120, valida tal abordagem. O tamanho do conjunto de dados pode ser bem grande.

Dessa forma, considera-se o quociente da divisão entre o número total de objetos e o número de linhas na submatriz (no exemplo, é 120), para saber o número de submatrizes necessário para representar o conjunto de dados completo. Se o quociente não for um valor inteiro, precisa-se arredondar para o número inteiro acima.

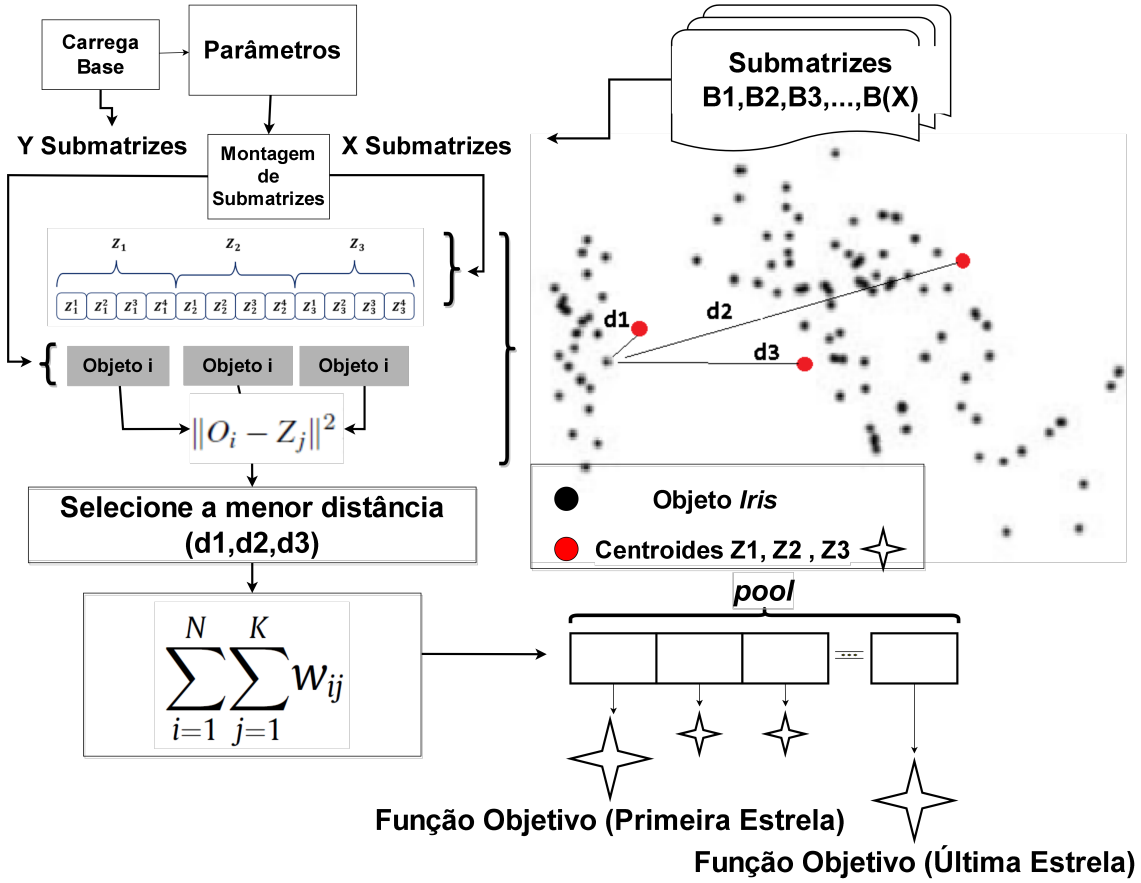
A parte inferior da Figura 8 mostra a função *Accel* em um formato mais geral para demonstrar a flexibilidade da abordagem na adaptação de outras funções objetivas. Neste trabalho, utilizou-se a função objetiva representada pela equação 1.

Figura 5 – Diagrama de montagem das submatrizes.



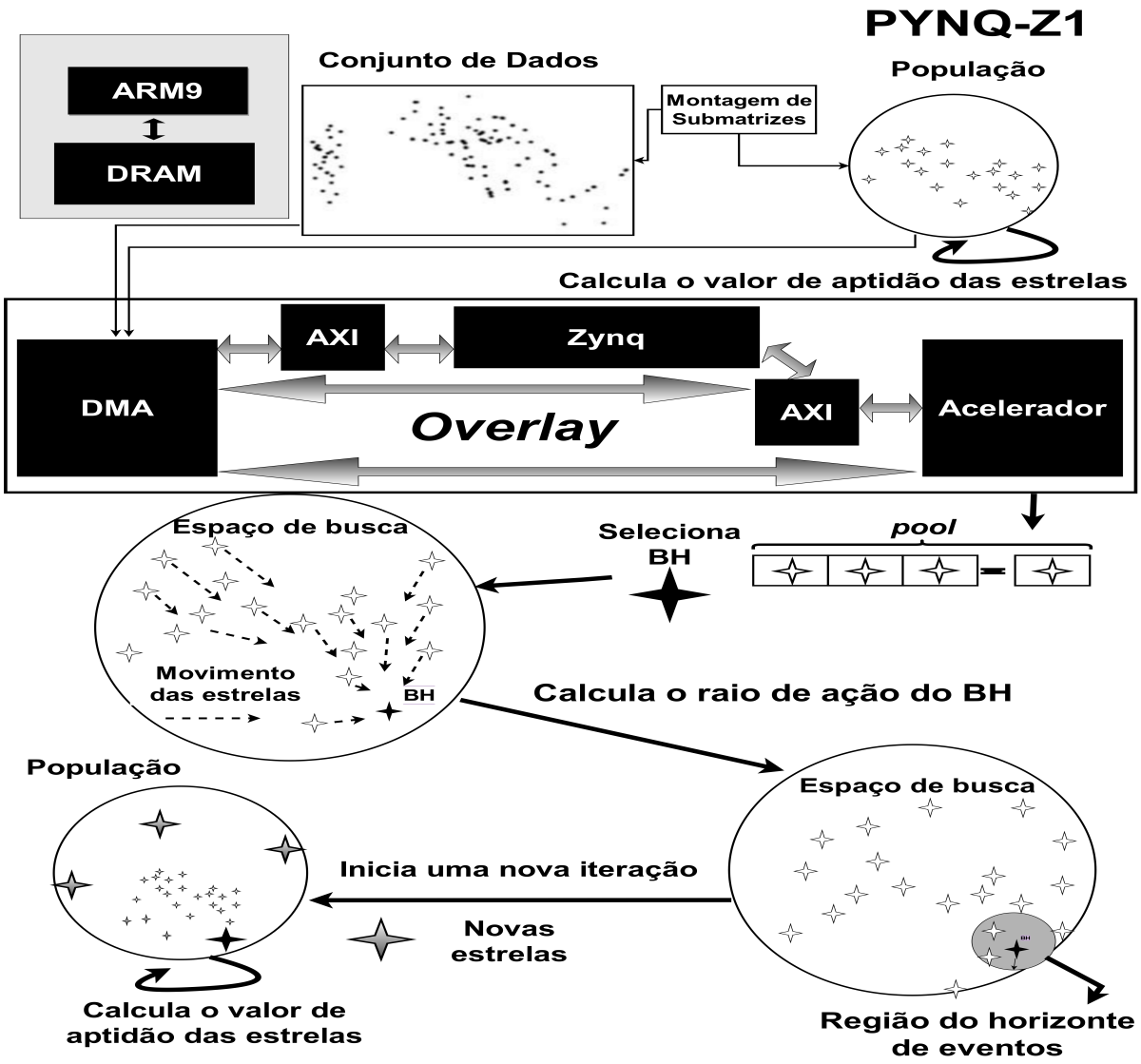
Fonte: Elaborado pelo autor (2022)

Figura 6 – Mecanismo de cálculo da função objetiva. O conjunto de dados *Iris* foi usado, $k = 3$ e $N = 150$. Os operadores e expressões são da equação 1 (exemplo).



Fonte: Elaborado pelo autor (2022)

Figura 7 – Diagrama geral da solução proposta.



Fonte: Produzida pelo autor

Figura 8 – Algoritmo do Black Hole. Eq. 20 representa o movimento das estrelas em direção ao buraco negro; Eq. 19 representa o raio de ação do BH; f_{BH} é o valor da função objetivo do buraco negro; X_{BH} é a posição do buraco negro; e $t = 1, 2, ...\ maximo\ nmero\ de\ iteraes$. A função *Accel* refere-se à parte acelerada pelo hardware, conforme explicado nesta seção.

Black Hole Algorithm

Input: número de estrelas (***N***), número máximo de iterações
Output: Black hole, valor da função objetiva do BH

Load(dataset) **%** carrega o conjunto de dados
A,*B* = **Assembly**(dataset,*N*) %Gera população de estrelas. Gera *Y* submatrizes *A* e *X* submatrizes *B*.
Begin
pool = ***Accel***(*A*,*B*) % calcula a função objetiva das estrelas e salva os resultados no vetor *pool*.
BH, X_{BH} = ***min***(*pool*) %A estrela com melhor custo é escolhida como **BH**
While (*Contagem de iterações máximas ou critério de parada*) **do**
For *i* = 1 to *números de estrelas*

$$x_i(t+1) = x_i(t) + rand \times (x_{BH} - x_i(t)) \tag{20}$$

Next *i*
B = ***update***(*B*) %atualiza as submatrizes *B*
pool = ***Accel***(*A*,*B*)
BH, X_{BH} = ***min***(*pool*)
For *i* = 1 **to** *número de estrelas*
If *o valor da função objetiva de* X_i < *o valor da função objetiva de* X_{BH} **Then**
$X_{BH} = X_i$
End if
Next *i*
Calcula o raio de ação do horizonte de eventos

$$R = \frac{f_{BH}}{\sum_{i=1}^{N} f_i} \tag{19}$$

For *i* = 1 **to** *número de estrelas*
If $\sqrt{(X_{BH} - X_i)^2} < R$ **Then**
substitua X_i por uma nova estrela em um lugar aleatório no espaço de busca
B = ***update***(*B*) %atualiza as submatrizes *B*
End if
Next *i*
End While
End Begin

Function Accel

Input: submatrizes (***A***) do conjunto de dados submatrizes (***B***) da população de estrelas
Output: ***pool*** (Valores da função objetiva das estrelas da população)

Begin
For *i* = 1 **to** *número de submatrizes* ***B***
For *j* =1 **to** *número das submatrizes* ***A***
pool = Função Objetiva (A,B) (1)
End
End
End
Return ***pool***

Fonte: Adaptada da referência (PASHAEI; AYDIN, 2017)

Capítulo 4

Resultados e Discussões

4.1 Testes dos Algoritmos BH, PSO, BB-BC e GSA para Agrupamento de Dados

Para a implementação de um sistema híbrido, inicialmente foi realizada uma investigação tendo em vista conhecer os algoritmos quanto às suas características, tamanho de população, número de iterações e tempo de execução. Uma vez que, na literatura, não foram encontradas informações sobre os tempos de execução de cada solução (HATAMLOU; ABDULLAH; NEZAMABADI-POUR, 2011; ESKANDARZADEHALAMDARY; MASOUMI; SOJODISHIJANI, 2014; BIJARI et al., 2018; HAN et al., 2017; DOWLATSHAHI; NEZAMABADI-POUR, 2014; SAHOO et al., 2014) justificou-se a replicação, no presente trabalho, para obter a validação temporal e para efetuar uma comparação mais justa.

Assim, foi proposta a hibridização e a paralelização do BH por meio de uma ferramenta de implementação de hardware de alto nível de última geração. Os algoritmos selecionados foram: *Particle Swarm Optimization* (PSO), *Gravitational Search Algorithm* (GSA) e *Big Bang - Big Crunch* (BB-BC). Os testes relacionados aos algoritmos selecionados foram realizados utilizando:

- ❏ Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz;
- ❏ 8,00 GB RAM;
- ❏ Windows 10;
- ❏ Matlab R2016a.

Os primeiros testes realizados neste trabalho avaliaram o desempenho dos algoritmos implementados e serviram para compará-los por meio de três critérios:

1. 1. A soma das distâncias euclidianas intragrupos (Eq. 1) como uma medida de qualidade interna dos resultados. Essa soma das distâncias intragrupos avaliou o melhor valor de aptidão; quanto menor for a soma, melhor será a posição dos centroides.

2. 2. A porcentagem de erro de objetos agrupados foi calculada e tratada como uma medida de qualidade externa. O cálculo da porcentagem foi realizado de acordo com a Equação 21.

$$ER = \frac{Nmero\ de\ objetos\ errados}{total\ de\ objetos} X100 \tag{21}$$

3. 3. O tempo médio de execução de cada algoritmo para processar um determinado número de iterações. Para todos os testes, foram feitas 50 simulações independentes, de modo a obter o valor médio, sendo utilizada uma população de 200 candidatos.

4.1.1 Estudo da soma das distâncias euclidianas intragrupos

A Tabela 3 mostra as menores somas das distâncias intragrupos obtidas pelos algoritmos de agrupamento de dados. Os valores indicados na tabela são: "melhor", "média", "pior"e "desvio padrão", obtidos após 50 simulações independentes. Para analisar a resposta, neste trabalho, dados extraídos da literatura (HATAMLOU, 2013) foram adicionados à tabela em uma coluna indicada por "*BH**". Todos os testes foram realizados com uma população de 200 candidatos.

Cada conjunto de dados teve um valor de iteração, sendo consideradas 10.000 iterações para o conjunto de dados *Iris*, 7.000 para o conjunto de dados *Wine* e 20.000 para os conjuntos de dados *Glass* e *CMC*. Embora não tenhamos os parâmetros de tamanho de população e número máximo de iterações utilizadas no trabalho de Hatamlou (HATAMLOU, 2013), essa comparação pode ser feita para avaliarmos se os nossos resultados foram satisfatórios.

Com os dados apresentados, foi realizada uma avaliação das respostas obtidas, sendo estas comparadas aos dados já descritos na literatura (BH*). Para o conjunto de dados *Iris*, todos os resultados obtidos pelo algoritmo BH foram melhores ou muito próximos aos demais algoritmos: PSO, BB-BC e GSA. O mesmo foi observado pelos valores descritos na literatura (HATAMLOU; ABDULLAH; HATAMLOU, 2011; HATAMLOU, 2013; NIKBAKHT; MIRVAZIRI, 2015).

Observação idêntica foi obtida com os resultados dos algoritmos PSO e BB-BC, nos quais o critério "melhor valor"foi ligeiramente superior ao descrito na literatura. Em relação ao algoritmo GSA, os resultados foram muito próximos, porém o GSA obteve

Tabela 3 – As menores somas das distâncias intragrupos obtidas por algoritmos implementados em Matlab e aplicados a diferentes conjuntos de dados. Para o algoritmo PSO foram considerados, $w = 0,51$, $d = 0,87$, $c1 = 1,37$ e $c2 = 1,01$; para o algoritmo BB-BC, $beta = 0,2$ e $alpha = 1$; e para o algoritmo GSA, $alpha = 20$ e $G_0 = 100$.

		Algoritmos				
Base	**Critério**	**BH***	**BH**	**BB-BC**	**PSO**	**GSA**
Iris	**Melhor**	96,66	**96,54**	**96,54**	**96,54**	**96,54**
	Média	96,66	**96,54**	104,83	97,37	**96,54**
	Pior	96,66	**96,54**	127,57	120,87	**96,54**
	Desvio Padrão	1,73E-03	**4,34E-04**	1,35E+01	3,43E+00	**2,06E-14**
Wine	**Melhor**	16.293,42	**16.292,00**	16.313,00	16.305,67	16.292,18
	Média	16.294,32	**16.292,00**	16.344,00	16.321,16	16.292,26
	Pior	16.300,23	**16.294,25**	16.411,61	16.360,96	16.297,49
	Desvio Padrão	1,65E+00	**3,40E-01**	2,11E+01	1,14E+01	7,23E-01
Glass	**Melhor**	210,52	**234,04**	246,21	255,25	**224,21**
	Média	211,5	**242,78**	272,42	292,57	**237,82**
	Pior	213,96	**258,92**	303,89	319,85	**265,06**
	Desvio Padrão	1,18E+00	**7,50E+00**	1,58E+01	1,28E+01	2,36E+01
CMC	**Melhor**	5.532,88	5.543,60	5.532,90	5.605,20	5.532,18
	Média	5.533,63	**5.546,30**	5.562,20	5.682,30	**5.532,18**
	Pior	5.534,78	5.548,70	5.723,05	5.808,88	5.532,18
	Desvio Padrão	5,99E-01	**8,70E-01**	3,01E+01	5,33E+01	**1,28E-12**

Os valores de *BH** foram extraídos da literatura (HATAMLOU, 2013).

um desvio padrão menor em relação ao desvio padrão do algoritmo BH tanto nos dados obtidos neste trabalho, quanto nos dados da literatura.

Avaliando o desempenho do algoritmo BH, considerando os conjuntos de dados testados, percebe-se que esse algoritmo teve seu melhor desempenho no conjunto de dados *Wine*.

Com relação à busca da menor soma das distâncias intragrupos, o algoritmo BH obteve os melhores resultados em relação aos algoritmos PSO, BB-BC, GSA, superando, também, os valores de referência. Os valores obtidos pelo BH para "melhor", "média", "pior" e "desvio padrão" foram 16.292,00, 16.292,00, 16.294,25 e 0,34, respectivamente.

É possível notar também que o pior valor ótimo do BH é numericamente melhor que os melhores resultados do PSO e BB-BC, que foram 16.305,67 e 16.313,00, respectivamente. Além disso, os valores médios da literatura (BH*) e do GSA (16.294,32 e 16.294,26, respectivamente) também são muito próximos do pior valor obtido pelo BH.

Para o conjunto *Glass*, o BH atingiu qualidade inferior ao GSA em relação aos valores "médio" e "melhor", mas obteve resultados melhores em relação ao pior valor obtido e desvio padrão. Os resultados dos algoritmos PSO e BB-BC foram os piores para os mesmos testes. O pior desempenho do BH foi para o conjunto *CMC*.

Como destaque positivo, os valores "médio", "pior" e "desvio padrão" foram superiores

aos resultados do algoritmo PSO e BB-BC, porém foram inferiores aos resultados do GSA e valores de referência da literatura. O mal desempenho do algoritmo BH, diante dos conjuntos *Iris* e *CMC*, deve ter ocorrido pela necessidade de ter um tamanho de população maior e/ou número máximo de iterações. Como não era objetivo deste estudo a otimização desses parâmetros, não foram realizados mais testes.

As Tabelas 4, 5, 6 e 7 mostram os melhores centroides obtidos pelo algoritmo BH nos conjuntos de dados *Iris*, *Wine*, *Glass* e *CMC*, respectivamente.

Tabela 4 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Iris*.

Centroide 1	Centroide 2	Centroide 3
6,73326425	5,014795908	5,934323335
3,067884103	3,41797216	2,798119365
5,630037761	1,468768373	4,417871544
2,106087826	0,2398792129	1,417352958

Tabela 5 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Wine*.

Centroide 1	Centroide 2	Centroide 3
12,53321286	12,81255311	13,71809075
2,331989741	2,549839977	1,889235709
2,327747672	2,370557358	2,410523319
21,32514423	19,50685713	16,92261692
92,53320821	98,94078914	105,2798053
2,049219832	2,076966115	2,836212085
1,790424259	1,510037851	3,057087959
0,3992191996	0,4002968582	0,3781410693
1,456253769	1,43660071	2,01467087
4,360937893	5,781257249	5,702056213
1,006459227	0,9542847599	1,084261105
2,468538317	2,240807506	2,997162263
463,5991462	686,9654483	1137,272126

Tabela 6 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *Glass*.

Centroide 1	Centroide 2	Centroide 3	Centroide 4	Centroide 5	Centroide 6
1,537515266	1,544355296	1,536880494	1,537215047	1,536851479	1,537595501
13,77532972	12,22981641	13,06401391	14,59087421	13,21977642	13,73046052
2,369848549	0,6338886523	3,501398395	0,4015279858	0,7020932498	3,410394818
2,517396668	1,340555343	1,399397545	2,162075357	1,546601613	1,190418478
71,16758445	72,05266689	72,90199295	73,26869208	72,99686743	71,94967143
2,648519157	0,763146503	0,6241460849	0,4085639017	0,7051213832	0,523275679
6,377067936	14,10548418	8,378658025	8,707513172	11,14412584	9,271980722
1,468340679	0,8921989498	0,2370373351	1,091039608	0,4684487766	0,355852263
0,2265219377	0,2193737353	0,1932666765	0,2038366017	0,251450933	0,2117215627

Tabela 7 – Os melhores centroides obtidos pelo algoritmo BH no conjunto de dados *CMC*.

Centroide 1	Centroide 2	Centroide 3
43,62836633	24,41697291	33,49191372
2,978358667	3,017623434	3,110597263
3,407052806	3,460352393	3,498969219
4,588585361	1,804933738	3,6541117
0,6899192949	0,7816693317	0,6986170916
0,6838394558	0,7005869717	0,6384155807
1,853717776	2,314279379	2,107490418
3,394036736	2,95364359	3,253721378
0,2244552569	0,213205441	0,2151218106

Os melhores centroides são apresentados para validar a soma das distâncias intragrupos na Tabela 3. Ao atribuir os centroides obtidos de cada conjunto de dados apresentados nas Tabelas 4, 5, 6 e 7, os melhores valores na Tabela 3 devem ser alcançados. Por exemplo, analisando o conjunto *Iris*, a soma das distâncias intragrupos deve ser alcançada (96,54). Caso contrário, os melhores valores na Tabela 3 ou os melhores centroides na Tabela 4 ou até mesmo ambos estão errados. Esse procedimento também pode ser feito para todos os conjuntos de dados deste trabalho.

4.1.2 Estudo da porcentagem de erro de agrupamento de dados

Os melhores centroides das Tabelas 4, 5, 6 e 7 foram também usados para calcular as taxas de erro de cada algoritmo testado para cada conjunto de dados. A Tabela 8 apresenta os resultados obtidos. Para os conjuntos *Iris* e *Wine*, nos quais o algoritmo BH atingiu os valores considerados ótimos globais, os resultados das taxas de erro se apresentaram discretamente melhores que os valores de referência. Entretanto, para os conjuntos *Glass* e *CMC*, o algoritmo BH não convergiu para os valores de referência e, consequentemente, os resultados das taxas de erro também não atingiram tais valores; porém, resultou muito próximo dos resultados dos demais algoritmos.

Quanto ao GSA, que havia apresentado o melhor desempenho de acordo com os critérios da Tabela 3, quando avaliado em relação à taxa de erro, gerou resultados que não foram tão satisfatórios para os conjuntos *Glass* e *CMC*. Isso mostra que um estudo mais profundo é importante na avaliação de desempenho de um algoritmo.

Para validar os critérios 1 e 2 das subseções (4.1.1 e 4.1.2) em relação às diferenças significativas entre os resultados obtidos pelos algoritmos de agrupamento, foram realizadas análises estatísticas. De acordo com Derrac et al. (DERRAC et al., 2011), o teste de *Friedman* é adequado para calcular a classificação média de quatro soluções algorítmicas, resolvendo quatro problemas de otimização. Para solução média e taxa de erro, foram calculadas as classificações médias dos algoritmos BH, PSO, BB-BC e GSA.

Os cálculos foram realizados no Matlab, no qual foi inserida uma tabela com todos

os resultados dos algoritmos estudados, aplicando a função *Friedman*. Os resultados são apresentados na Tabela 9.

O nível de confiança $\alpha = 0{,}05$ foi usado em todos os casos. A classificação média dos algoritmos de agrupamento foi obtida pelo teste de *Friedman* com base na soma das médias das distâncias intragrupos. O algoritmo GSA é classificado em primeiro lugar com $rank = 1{,}25$, seguido pelo algoritmo BH com $rank = 1{,}75$ e os algoritmos PSO e BB-BC, ambos com 3,5. Para o teste de taxas de erro, o valor α resultante ficou acima do nível de confiança esperado e, portanto, não houve diferenças significativas entre os resultados da medida de qualidade externa. Em seguida, novos testes foram realizados para manter a mesma metodologia utilizada por Hatamlou (HATAMLOU, 2013), no qual foi utilizado o teste de *Friedman*, seguido do teste de *Iman-Davenport* e um pós-teste de *Holms*.

O *p-value* calculado pelo teste de *Friedman* e o teste de *Iman-Davenport* estão na Tabela 10. Ambos rejeitam a hipótese nula de desempenho equivalente e confirmam a existência de diferenças significativas entre o desempenho de todos os algoritmos de agrupamento. Portanto, o método de *Holm* é um pós-teste para detectar diferenças estatísticas efetivas entre os algoritmos com o menor *rank* de *Friedman*. Os resultados do método de *Holm* revelaram que não há diferença significativa com base nos resultados entre o algoritmo BH e o algoritmo GSA, e ambos são superiores aos algoritmos PSO e BB-BC.

A realização dos testes estatísticos não paramétricos utilizados neste trabalho foi apoiada pela literatura (DERRAC et al., 2011; ABDI, 2010) para manter uma comparação mais justa com o trabalho de Hatamlou (HATAMLOU, 2013). Assim, foi considerado um cenário com quatro problemas (conjunto de dados), avaliando, desta forma, quatro algoritmos. Foi utilizado o teste de *Friedman*, que nos permitiu distinguir alguns problemas seguindo um determinado critério. Os fatores de tempo de resposta e simplicidade de implementação, que não fazem parte dos testes estatísticos, sustentam a proposta de otimização do algoritmo BH, embora o algoritmo GSA tenha apresentado melhor desempenho em alguns casos.

Tabela 8 – A taxa de erro (%) dos algoritmos testados. **BH*** representa os valores extraídos da literatura (HATAMLOU, 2013).

Conjunto de Dados	**BH***	**BH**	**BB-BC**	**PSO**	**GSA**
Iris	10,02	10,00	17,85	10,25	10,00
Wine	28,47	28,09	28,98	28,70	29,17
Glass	36,51	42,99	41,37	41,20	41,39
CMC	54,39	57,16	57,05	56,85	57,16

Tabela 9 – Classificação média obtida pelo teste de *Friedman* utilizando os parâmetros: "valor médio da soma das distâncias intragrupos" e "taxas de erro".

Classificação média	*p-value*	BH	PSO	BB-BC	GSA
Soma das distâncias intragrupos	0,02	1,75	3,50	3,50	1,25
Taxas de erro	0,66	2,38	2,00	2,50	3,13

4.1.3 Estudo do tempo de execução dos algoritmos

Como já mencionado anteriormente, o algoritmo BH se mostrou melhor que os demais algoritmos para resolver o mesmo conjunto de dados (HATAMLOU, 2013). Entretanto, o estudo e a implementação das soluções permitiram identificar, em cada algoritmo, etapas que exigiram mais computação. Analisando os passos de cada algoritmo, percebeu-se a simplicidade do algoritmo BH em relação aos demais algoritmos. Na busca por uma solução ótima, o algoritmo BH utilizou apenas as informações de localização da sua população e do Buraco Negro como mecanismo de busca da melhor solução.

Esse mecanismo é representado pelo movimento das estrelas em direção ao buraco negro (Eq. 20). Já o PSO tem uma etapa adicional em seu mecanismo de busca representado pela Equação 4. Essa etapa adicional consiste no cálculo da velocidade de cada candidato (Equação 2). Nessa mesma relação, o GSA tem três etapas a mais a serem processadas, se comparado ao algoritmo BH: velocidade (Equação 16), aceleração (Equação 15) e força de cada candidato na população (Equação 13). Além disso, o algoritmo BH é livre de parâmetros, o que pode ser considerado uma vantagem em relação aos demais algoritmos devido à sua simplicidade de implementação e sua independência de variáveis para resolver problemas de uso geral, justificando seu uso em projetos de hardware e aplicativos SoC.

Para se chegar à análise do tempo de resposta de cada algoritmo, a Figura 9 mostra o comportamento dos algoritmos e as soluções obtidas a cada iteração. O tempo médio de execução para resolver cada conjunto de dados também é mostrado. A primeira coluna mostra quatro gráficos com os melhores resultados encontrados em cada iteração, um para cada conjunto de dados. Uma linha roxa destaca o valor de referência da literatura (HATAMLOU, 2013).

Para todos os algoritmos, foi utilizada uma amostra de 2000 iterações. O primeiro

Tabela 10 – Resultados dos testes de *Friedman* e *Iman-Davenport* com base na soma das distâncias intragrupos

Método	*p-value*	*Valor estatístico*	Hipótese
Friedman	0,02	9,90	Rejeitado
Iman-Davenport	0,001	14,14	Rejeitado

Figura 9 – Testes dos Algoritmos BH, PSO, BB-BC e GSA, implementados em Matlab. Para o algoritmo PSO, $w = 0,51$, $d = 0,87$, $c1 = 1,37$ e $c2 = 1,01$ foram considerados; para o algoritmo BB-BC, $beta = 0,2$ e $alpha = 1$; e para o algoritmo GSA, $alpha = 20$ e $G_0 = 100$.

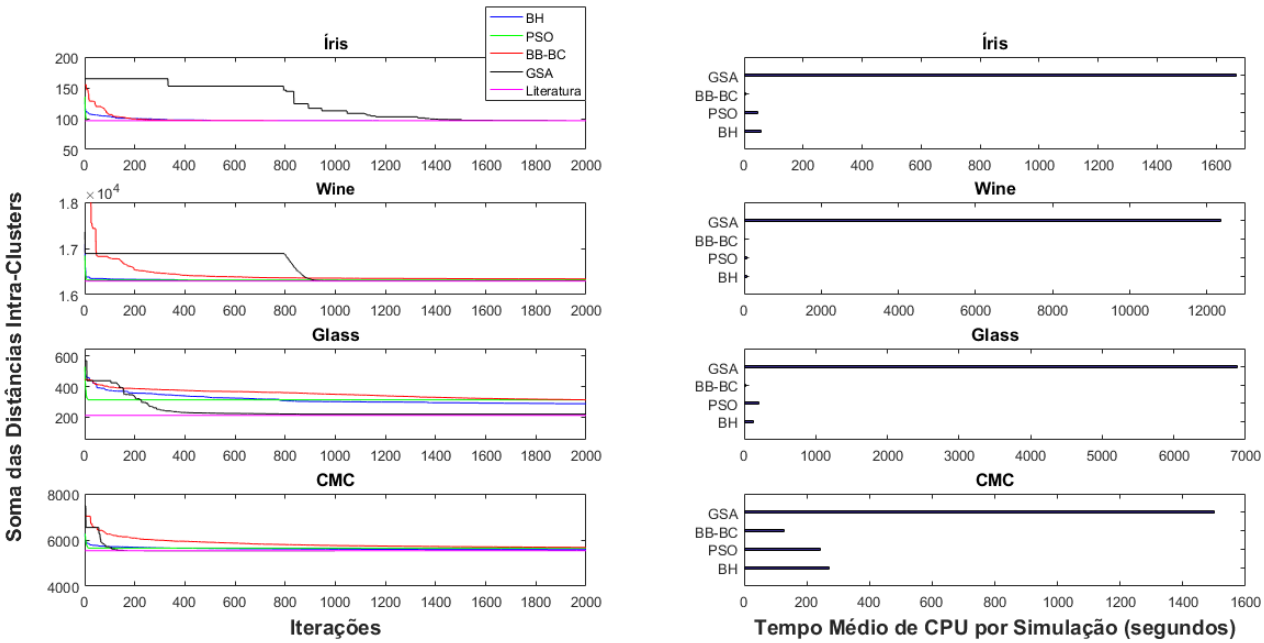


gráfico mostra o desempenho dos algoritmos na resolução do agrupamento do conjunto *Iris*. Os algoritmos BH, PSO e BB-BC convergiram para valores próximos ao ótimo global após as primeiras 200 iterações, enquanto o GSA precisou de mais de 1400 iterações para se aproximar dos valores alcançados pelos outros algoritmos. Nos conjuntos *Wine* e *CMC*, todos os algoritmos convergiram para valores próximos ao valor ótimo global, na proximidade da milésima iteração. O gráfico com os resultados para o conjunto de dados *Glass* mostra um pior desempenho do BH nas primeiras 2000 iterações quando comparado ao algoritmo GSA.

Os quatro gráficos na segunda coluna apresentam o tempo médio, em segundos, que cada algoritmo levou para executar a mesma amostra de 2000 iterações para cada conjunto de dados. O primeiro gráfico da segunda coluna mostra que o GSA foi muito mais lento em comparação aos outros algoritmos. Esse mesmo comportamento foi observado para todos os conjuntos de dados, conforme mostrado nos gráficos da segunda coluna.

Mesmo com um tamanho de amostra relativamente pequeno, o algoritmo GSA claramente tem o pior desempenho de tempo de execução quando comparado aos outros algoritmos. Embora o algoritmo BH não tenha obtido a melhor performance em relação aos algoritmos PSO e BB-BC, em termos de tempo de resposta, conforme resultados analisados nas sub-seções 4.1.1 e 4.1.2, o algoritmo BH concorre com o algoritmo GSA.

Assim, inferiu-se que o algoritmo BH é mais vantajoso para proposta deste trabalho.

4.2 Estudo do tempo de execução do algoritmo BH implementado na Pynq-Z1

Em geral, de acordo com as amostras utilizadas, o BH obteve qualidade próxima ao GSA, sem diferenças significativas na taxa de erro, e foi superior ao algoritmo GSA em termos de tempo de resposta da solução. Essa observação apoia a proposta de implementação de um algoritmo simples, eficiente, eficaz e livre de parâmetros por meio de uma solução de hardware híbrido para melhorar o *speedup*, medida de desempenho relativa de dois sistemas processando o mesmo problema.

Para isso, duas amostras foram comparadas: i) o algoritmo BH implementado na Pynq-Z1 usando apenas o processamento em nível de software (ou seja, sem utilização do acelerador em hardware); e ii) o algoritmo BH implementado de forma híbrida na Pynq-Z1, utilizando o acelerador em hardware desenvolvido.

A amostra I, que foi chamada de "*BH SW*", utiliza um bloco de código implementado para executar em software a mesma função do acelerador em hardware. O bloco de código utilizado é mostrado no formato de um pseudo-código e está na Figura 10.

A amostra II, que foi chamada de "*BH HW*", utiliza o acelerador em hardware. Para cada conjunto de dados, as duas amostras foram executadas utilizando os seguintes parâmetros: 100 simulações independentes do algoritmo BH, executando apenas uma iteração, sendo considerada uma população de 200 soluções candidatas. Os resultados do tempo médio são apresentados na Figura 11 e na Tabela 11.

A Tabela 11 mostra que a solução proposta obteve um *speedup* de mais de 30 vezes para o conjunto de dados *Iris*, se comparada a uma solução totalmente executada no nível do software. O *speedup* mais baixo foi para o conjunto de dados *Wine* e, mesmo assim, foi na ordem de 15 vezes mais rápido que o *BH SW*. Já para o *Glass* e *CMC*, o *speedup* da solução proposta foi de 24,63 e 27,08 vezes, respectivamente. A Figura 11 mostra os mesmos resultados em gráfico de barras para uma melhor visualização.

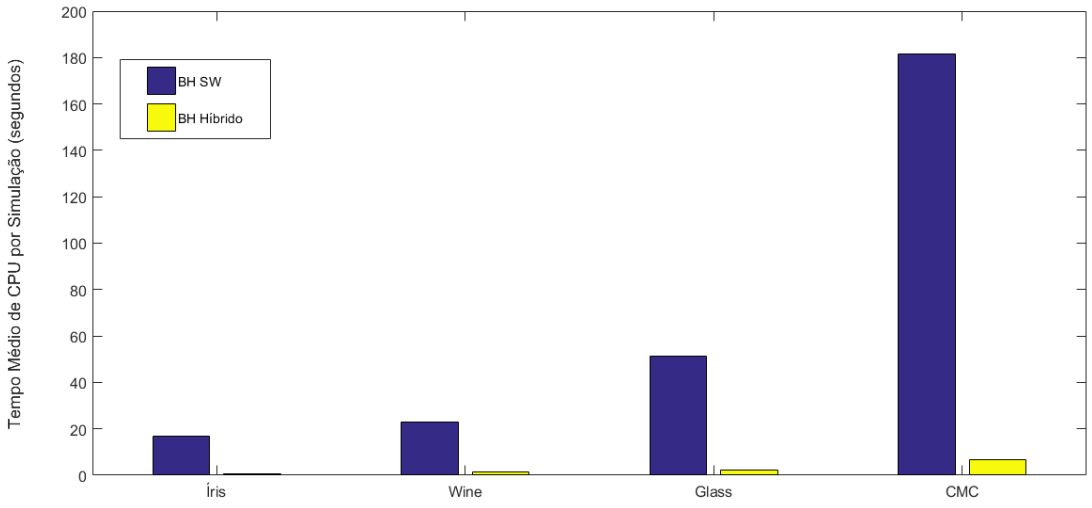
Tabela 11 – Tempo médio de execução para uma iteração com mais de 100 simulações independentes.

Conjunto de Dados	*BH SW* (segundos)	*BH Híbrido* (segundos)
Iris	16,71	0,55
Wine	22,92	1,49
Glass	51,29	2,08
CMC	181,54	6,70

Figura 10 – Pseudo-código da função objetiva utilizada na amostra *BH SW*

for $i = 1, 2, \ldots, Número\ de\ Candidatos\ a\ Solução$ **do**
  $fitness = 0$
  for $j = 1, 2, \ldots, Número\ de\ Objetos\ do\ Conjunto\ de\ Dados$ **do**
    $distances = 0$
    for $k = 1, 2, \ldots, Number\ of\ Centroids$ **do**
      $distances = sqrt(sum(star_i(Z_k) - object_j)^2)$
    end for
    $fitness = fitness + minimum(distances)$
  end for
  $pool(i) = fitness$
end for

Figura 11 – Tempo médio de execução para uma iteração com mais de 100 simulações independentes.



Capítulo 5

Conclusão

O objetivo principal neste trabalho foi alcançado, sendo realizada a hibridização e a paralelização do algoritmo BH em um SoC para resolver a tarefa de agrupamento de dados.

Foram implementados os algoritmos BH, PSO, BB-BC e GSA no software Matlab, os quais permitiram realizar testes comparativos entre os algoritmos aplicados na tarefa de agrupamento de quatro conjuntos de dados. Esses testes, realizados de forma empírica, auxiliaram na busca de amostras satisfatórias em relação ao tamanho da população e ao número de iterações que cada algoritmo utilizou para convergir para valores próximos aos pontos ótimos globais conhecidos na literatura.

Além disso, foram investigados os parâmetros temporais dos experimentos, como realizado por outros autores na literatura, o que favoreceu uma comparação mais justa. A medida de qualidade para a avaliação foi parametrizada de duas formas: pela soma das distâncias intragrupos, como uma medida de qualidade interna, e a taxa de erro de agrupamento realizado pelas melhores soluções obtidas, como uma medida de qualidade externa.

Os resultados das soluções obtidas pelos algoritmos implementados foram avaliados utilizando três testes estatísticos: teste de *Friedman*, teste *Iman-Davenport* e um pós-teste de *Holm*. Com esses testes, foi observado que não havia diferenças significativas nas soluções obtidas pelos algoritmos BH e GSA, e que ambos foram superiores aos algoritmos PSO e BB-BC.

Ao final desses testes, pode-se reconhecer as principais características atribuídas ao algoritmo BH: simplicidade e facilidade de implementação, livre parametrização, flexibilidade a modificações e bom desempenho para a tarefa de agrupamento de dados.

O algoritmo BH teve um bom desempenho em termos de tempo de execução, em re-

lação aos outros algoritmos; principalmente quando comparado ao algoritmo GSA, que obteve resultados com qualidade bem competitiva em relação ao BH. Contudo, com desempenho muito inferior na questão temporal.

Para a implementação da solução de hibridização foi utilizado o Vivado HLS, ferramenta de implementação de hardware de alto nível, de última geração. Essa solução permite que qualquer usuário não especialista em hardware implemente hardware dedicado em considerado período curto de tempo, quando comparado a um método mais tradicional, como o Verilog-HDL ou VHDL.

Dessa forma, utilizando a linguagem C++, foi implementado um núcleo IP dedicado para calcular a função objetiva de uma população diante de um determinado conjunto de dados. A representação dos dados e a maneira como foi projetado o fluxo de dados consubstanciaram técnicas que permitiram explorar o paralelismo nas partes mais custosas da solução.

Embora tenha sido necessário sintetizar um acelerador para cada conjunto de dados devido às diferentes características de cada conjunto de dados utilizado, a abordagem de particionar a representação dos dados em matrizes menores e de tamanho fixo mostrou-se flexível a adaptações, e escalável ante limitações de hardware, de tal forma que essa abordagem pode ser utilizada com outros algoritmos e conjunto de dados maiores e mais complexos.

Ao final da implementação proposta, demonstrou-se que o melhor tempo de execução foi obtido com o novo procedimento em hardware, contra a mesma versão em software. Por fim, a proposta alcançou *speedup* médio de 25 vezes quando comparada à mesma solução em software.

Referências

ABDI, H. Holms sequential bonferroni procedure. **Encyclopedia of research design**, v. 1, n. 8, p. 1–8, 2010.

ALIPPI, C. et al. Energy management in wireless sensor networks with energy-hungry sensors. **IEEE Instrumentation & Measurement Magazine**, IEEE, v. 12, n. 2, p. 16–23, 2009.

BAE, W. Todays computing challenges: opportunities for computer hardware design. **PeerJ Computer Science**, PeerJ Inc., v. 7, p. e420, 2021.

BAGNI, D. et al. A zynq accelerator for floating point matrix multiplication designed with vivado hls. **Application note, January**, 2016.

BASU, A. et al. Low-power, adaptive neuromorphic systems: Recent progress and future directions. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, IEEE, v. 8, n. 1, p. 6–27, 2018.

BIJARI, K. et al. Memory-enriched big bang–big crunch optimization algorithm for data clustering. **Neural Computing and Applications**, Springer, v. 29, n. 6, p. 111–121, 2018.

BLAKE, C. L.; MERZ, C. J. **UCI repository of machine learning databases, 1998**. 1998. <http://archive.ics.uci.edu/ml/datasets.php>. Acesso em: 27 nov. 2016.

BOUVIER, M. et al. Spiking neural networks hardware implementations and challenges: A survey. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, ACM New York, NY, USA, v. 15, n. 2, p. 1–35, 2019.

CANIS, A. et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In: **Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2011. p. 33–36.

CASS, S. The 2018 top programming languages. **IEEE Spectrum**, v. 31, 2018.

CONG, J. et al. High-level synthesis for fpgas: From prototyping to deployment. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 30, n. 4, p. 473–491, 2011.

CORRADI, G. The value of python productivity: Extreme edge analytics on xilinx zynq portfolio. **PYNQ Framework and the Zynq Portfolio. Xilinx**, 2018.

DERRAC, J. et al. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. **Swarm and Evolutionary Computation**, Elsevier, v. 1, n. 1, p. 3–18, 2011.

DITZLER, G. et al. Learning in nonstationary environments: A survey. **IEEE Computational Intelligence Magazine**, IEEE, v. 10, n. 4, p. 12–25, 2015.

DOWLATSHAHI, M. B.; NEZAMABADI-POUR, H. Ggsa: a grouping gravitational search algorithm for data clustering. **Engineering Applications of Artificial Intelligence**, Elsevier, v. 36, p. 114–121, 2014.

DUA, D.; GRAFF, C. **UCI Machine Learning Repository**. 2017. Disponível em: <http://archive.ics.uci.edu/ml>.

ELAZAB, H.-A. I. et al. Fpga eco unit commitment based gravitational search algorithm integrating plug-in electric vehicles. **Energies**, Multidisciplinary Digital Publishing Institute, v. 11, n. 10, p. 2547, 2018.

EROL, O. K.; EKSIN, I. A new optimization method: big bang–big crunch. **Advances in Engineering Software**, Elsevier, v. 37, n. 2, p. 106–111, 2006.

ESKANDARZADEHALAMDARY, M.; MASOUMI, B.; SOJODISHIJANI, O. A new hybrid algorithm based on black hole optimization and bisecting k-means for cluster analysis. In: IEEE. **2014 22nd Iranian Conference on Electrical Engineering (ICEE)**. [S.l.], 2014. p. 1075–1079.

FILHO, A. G. S. et al. Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm. In: IEEE. **16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.** [S.l.], 2003. p. 99–104.

GIACCONI, R. Black hole research past and future. In: **Black Holes in Binaries and Galactic Nuclei: Diagnostics, Demography and Formation**. [S.l.]: Springer, 2001. p. 3–15.

GU, S.; HAO, T.; YANG, S. The implementation of a pointer network model for traveling salesman problem on a xilinx pynq board. In: SPRINGER. **International Symposium on Neural Networks**. [S.l.], 2018. p. 130–138.

HALLIDAY, D.; RESNICK, R.; WALKER, J. **Fundamentals of Physics (pp 300-301)**. [S.l.]: New York: John Wiley & Sons, inc, 1993.

HAN, X. et al. A novel data clustering algorithm based on modified gravitational search algorithm. **Engineering Applications of Artificial Intelligence**, Elsevier, v. 61, p. 1–7, 2017.

HATAMLOU, A. Black hole: A new heuristic optimization approach for data clustering. **Information sciences**, Elsevier, v. 222, p. 175–184, 2013.

HATAMLOU, A.; ABDULLAH, S.; HATAMLOU, M. Data clustering using big bang–big crunch algorithm. In: SPRINGER. **International Conference on Innovative Computing Technology**. [S.l.], 2011. p. 383–388.

HATAMLOU, A.; ABDULLAH, S.; NEZAMABADI-POUR, H. Application of gravitational search algorithm on data clustering. In: SPRINGER. **International conference on rough sets and knowledge technology**. [S.l.], 2011. p. 337–346.

HIGHAM DESMOND J E HIGHAM, N. J. **guia do MATLAB**. [S.l.: s.n.].

HONDA, K.; WEI, K.; AMANO, H. Fpga/python co-design for lane line detection on a pynq-z1 board. In: IEEE. **2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)**. [S.l.], 2019. p. 53–60.

IBRAHIM, Z. et al. A survey on applications of black hole algorithm. 2018.

JAIN, A. K. Data clustering: 50 years beyond k-means. **Pattern recognition letters**, Elsevier, v. 31, n. 8, p. 651–666, 2010.

JANSSEN, B.; ZIMPRICH, P.; HÜBNER, M. A dynamic partial reconfigurable overlay concept for pynq. In: IEEE. **2017 27th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.], 2017. p. 1–4.

JIANG, S.; WANG, Y.; JI, Z. Convergence analysis and performance of an improved gravitational search algorithm. **Applied Soft Computing**, Elsevier, v. 24, p. 363–384, 2014.

JUNCHAO, Z.; WEILIANG, C.; SHAOJUN, W. Parameterized ip core design. In: IEEE. **ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No. 01TH8549)**. [S.l.], 2001. p. 744–747.

"JUPYTER". **[Online]. "Jupyter Notebooks". Available:**. 2022. <https://www.jupyter.org>.

KAVEH, A.; TALATAHARI, S. Size optimization of space trusses using big bang–big crunch algorithm. **Computers & structures**, Elsevier, v. 87, n. 17-18, p. 1129–1140, 2009.

_____. Optimal design of schwedler and ribbed domes via hybrid big bang–big crunch algorithm. **Journal of Constructional Steel Research**, Elsevier, v. 66, n. 3, p. 412–419, 2010.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: IEEE. **Proceedings of ICNN'95-International Conference on Neural Networks**. [S.l.], 1995. v. 4, p. 1942–1948.

KUMAR, S.; DATTA, D.; SINGH, S. K. Black hole algorithm and its applications. In: **Computational intelligence applications in modeling and control**. [S.l.]: Springer, 2015. p. 147–170.

KUMBASAR, T. et al. Adaptive fuzzy model based inverse controller design using bb-bc optimization algorithm. **Expert Systems with Applications**, Elsevier, v. 38, n. 10, p. 12356–12364, 2011.

LIU, S.-C. et al. **Event-based neuromorphic systems**. [S.l.]: John Wiley & Sons, 2014.

LYSAGHT, P. Future design tools for platform fpgas. In: IEEE. **16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.** [S.l.], 2003. p. 275–280.

MAHDAD, B.; SRAIRI, K. Interactive gravitational search algorithm and pattern search algorithms for practical dynamic economic dispatch. **International Transactions on Electrical Energy Systems**, Wiley Online Library, v. 25, n. 10, p. 2289–2309, 2015.

MOLANES, R. F. et al. Deep learning and reconfigurable platforms in the internet of things: Challenges and opportunities in algorithms and hardware. **IEEE Industrial Electronics Magazine**, IEEE, v. 12, n. 2, p. 36–49, 2018.

NEMATI, M.; MOMENI, H.; BAZRKAR, N. Binary black holes algorithm. **International Journal of Computer Applications**, Citeseer, v. 79, n. 6, 2013.

NIKBAKHT, H.; MIRVAZIRI, H. A new algorithm for data clustering based on gravitational search algorithm and genetic operators. In: IEEE. **2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP)**. [S.l.], 2015. p. 222–227.

PASHAEI, E.; AYDIN, N. Binary black hole algorithm for feature selection and classification on biological data. **Applied Soft Computing**, Elsevier, v. 56, p. 94–106, 2017.

"PYNQ". **[Online]. Available:**. 2022. <http://www.pynq.io>.

RASHEDI, E.; NEZAMABADI-POUR, H.; SARYAZDI, S. Gsa: a gravitational search algorithm. **Information sciences**, Elsevier, v. 179, n. 13, p. 2232–2248, 2009.

RASHEDI, E.; RASHEDI, E.; NEZAMABADI-POUR, H. A comprehensive survey on gravitational search algorithm. **Swarm and evolutionary computation**, Elsevier, v. 41, p. 141–158, 2018.

SAEGUSA, T.; MARUYAMA, T. An fpga implementation of real-time k-means clustering for color images. **Journal of Real-Time Image Processing**, Springer, v. 2, n. 4, p. 309–318, 2007.

SAHOO, G. et al. A review on gravitational search algorithm and its applications to data clustering & classification. **IJ Intelligent Systems and Applications**, v. 6, p. 79–93, 2014.

SALVADOR, R. Evolvable hardware in fpgas: Embedded tutorial. In: IEEE. **2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)**. [S.l.], 2016. p. 1–6.

SKALICKY, S. et al. Hot & spicy: Improving productivity with python and hls for fpgas. In: IEEE. **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2018. p. 85–92.

SMITH, L. S. Neuromorphic systems: Past, present and future. **Brain Inspired Cognitive Systems 2008**, Springer, p. 167–182, 2010.

STORNAIUOLO, L.; SANTAMBROGIO, M.; SCIUTO, D. On how to efficiently implement deep learning algorithms on pynq platform. In: IEEE. **2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.], 2018. p. 587–590.

SZE, V. et al. Hardware for machine learning: Challenges and opportunities. In: IEEE. **2017 IEEE Custom Integrated Circuits Conference (CICC)**. [S.l.], 2017. p. 1–8.

TANG, H. et al. Big bang-big crunch optimization for parameter estimation in structural systems. **Mechanical Systems and Signal Processing**, Elsevier, v. 24, n. 8, p. 2888–2897, 2010.

TREFZER, M. A.; TYRRELL, A. M. Evolvable hardware. **From Practice to Application. Springer**, Springer, 2015.

TSAI, C.-W.; HSIEH, C.-H.; CHIANG, M.-C. Parallel black hole clustering based on mapreduce. In: IEEE. **2015 IEEE International Conference on Systems, Man, and Cybernetics**. [S.l.], 2015. p. 2543–2548.

VANROSSUM, G.; DRAKE, F. L. **The python language reference**. [S.l.]: Python Software Foundation Amsterdam, Netherlands, 2010.

"VIVADO". **"Vivado Design Suite User Guide: High-Level Synthesis"**. 2020. <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug871-vivado-high-level-synthesis-tutorial.pdf>.

WANG, Y. et al. Kpynq: A work-efficient triangle-inequality based k-means on fpga. In: IEEE. **2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2019. p. 320–320.

WINTERSTEIN, F.; BAYLISS, S.; CONSTANTINIDES, G. A. High-level synthesis of dynamic data structures: A case study using vivado hls. In: IEEE. **2013 International Conference on Field-Programmable Technology (FPT)**. [S.l.], 2013. p. 362–365.

XILINX, I. Vivado design suite user guide. **High-Level Synthesis**, 2014.

APÊNDICE A

Código utilizado para o desenvolvimento do acelerador em hardware no software Vivado HLS

Todo código desenvolvido e apresentado neste Apêndice não foi otimizado. O objetivo é compartilhar os códigos que foram utilizados para extrair os resultados deste trabalho.

O software Vivado HLS na versão 2019v2 foi utilizado para desenvolver e sintetizar os 4 núcleos IP deste estudo, um para cada conjunto de dados. O modelo da plataforma utilizada foi: xc7z020clg400-1 para PYNQ Z1 FPGA.Dois arquivos foram utilizados: "*matmult_accel.cpp*"e "*matmult.h*". O código está na linguagem de programação C++, e estão apresentados nas Figuras 12 e 13. De acordo com a base de dados, respeitando as características de cada conjunto de dados, um bloco de código em específico, foi adicionado ao código da Figura 13. São eles:

- Para a base *Iris*, foi utilizado o código da Figura 14;
- Para a base *Wine*, foi utilizado o código da Figura 15;
- Para a base *Glass*, foi utilizado o código da Figura 16;
- Para a base *CMC*, foi utilizado o código da Figura 17.

Figura 12 – Arquivo "*matmult.h*". Código em C++ da Implementação do acelerador em hardware.

```
#ifndef _XF_AXIS_CONFIG_
#define _XF_AXIS_CONFIG_

#include "ap_int.h"
#include "hls_stream.h"
#include <iostream>
#include <math.h>
using namespace std;
#include <cmath>

#define N 120
#define SIZE 14400 // N*N

struct axis_t {
  ap_uint<32> data;
  ap_int<1> last;
};

template <typename T> void mmult_hw(T a[N][N], T b[N][N], T out[N][N]);

#endif
```

Fonte: Adaptada da referência (BAGNI et al., 2016).

Figura 13 – Arquivo "*matmult_accel.cpp*". Código em C++ da Implementação do acelerador em hardware.

```
template <typename T> void axis2Mat(axis_t *src, T A[N][N], T B[N][N]) {
#pragma HLS inline off
  union {    int ival; T oval;  } converter;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      int k = i * N + j;
      converter.ival = src[k].data;
      A[i][j] = converter.oval;}}
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      int k = i * N + j;
      converter.ival = src[k + SIZE].data;
      B[i][j] = converter.oval;}}
}
template <typename T> void Mat2axis(T C[N][N], axis_t *dst) {
#pragma HLS inline off
  union {    int oval; T ival;  } converter;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      ap_uint<1> tmp = 0;
      if ((i == N - 1) && (j == N - 1)) {
        tmp = 1;}
      dst[i * N + j].last = tmp;
      converter.ival = C[i][j];
      dst[i * N + j].data = converter.oval;}}
}

extern "C" {
void matmult_accel(axis_t *src, axis_t *dst) {
#pragma HLS INTERFACE axis port = src
#pragma HLS INTERFACE axis port = dst
#pragma HLS INTERFACE s_axilite port = return
#pragma HLS dataflow
  float A[N][N];
  float B[N][N];
  float C[N][N];
  axis2Mat(src, A, B);
  mmult_hw(A, B, C);
  Mat2axis(C, dst);}
}
```

Fonte: Adaptada da referência (BAGNI et al., 2016).

Figura 14 – Código em C++ da Implementação do acelerador em hardware para a base *Iris*

```
#include "matmult.h"

// function to be accelerated in HW
template <typename T> void mmult_hw(T a[N][N], T b[N][N], T out[N][N]) {
float res[N][N];
L1:
  for (int ia = 0; ia < N; ++ia)
  L2:
    for (int ib = 0; ib < N; ++ib) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      res[ia][ib] = ((a[ia][ib] - b[ia][ib])*(a[ia][ib] - b[ia][ib]));
    }
int nv =4;   //tamanho do objeto
int pop = 10;  //número de candidatos por submatriz
int k = 3;    //número de grupos
int nvk=nv*k;   //tamanho do candidato
float fitness[pop];
  for (int i =0; i < pop; ++i){

    for (int ia = 0; ia < N; ++ia){
        float sum1 = 0;
        float sum2 = 0;
        float sum3 = 0;
        int idx=i*nvk;
        for (int ib = 0; ib < nv; ++ib){
#pragma HLS pipeline
#pragma HLS loop_flatten off
            sum1 += res[ia][idx+ib];
            sum2 += res[ia][idx+ib+nv];
            sum3 += res[ia][idx+ib+nv*2];
        }

        float a = sqrt(sum1);
        float b = sqrt(sum2);
        float c = sqrt(sum3);
        float smaller = min(a,b);
        smaller = min(smaller,c);
      sum_t += smaller;}
      fitness[i] = sum_t;
    }
  for (int ic = 0; ic < pop; ++ic){
      out[0][ic]=fitness[ic];
  }
  return;
}
```

Fonte: Produzida pelo autor

Figura 15 – Código em C++ da Implementação do acelerador em hardware para a base *Wine*.

```
#include "matmult.h"

// function to be accelerated in HW
template <typename T> void mmult_hw(T a[N][N], T b[N][N], T out[N][N]) {
float res[N][N];
L1:
  for (int ia = 0; ia < N; ++ia)
  L2:
    for (int ib = 0; ib < N; ++ib) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      res[ia][ib] = ((a[ia][ib] - b[ia][ib])*(a[ia][ib] - b[ia][ib]));
    }
int nv =13;   //tamanho do objeto da base Wine
int pop = 3;  //número de candidatos por submatriz
int k = 3;    //número de grupos
int nvk=nv*k; //tamanho do candidato
float fitness[pop];
  for (int i =0; i < pop; ++i){
    float sum_t = 0;

    for (int ia = 0; ia < N; ++ia){
        float sum1 = 0;
        float sum2 = 0;
        float sum3 = 0;
        int idx=i*nvk;
        for (int ib = 0; ib < nv; ++ib){
#pragma HLS pipeline
#pragma HLS loop_flatten off
            sum1 += res[ia][idx+ib];
            sum2 += res[ia][idx+ib+nv];
            sum3 += res[ia][idx+ib+nv*2];
        }

        float a = sqrt(sum1);
        float b = sqrt(sum2);
        float c = sqrt(sum3);
        float smaller = min(a,b);
        smaller = min(smaller,c);
        sum_t += smaller;}
      fitness[i] = sum_t;
    }
  for (int ic = 0; ic < pop; ++ic){
      out[0][ic]=fitness[ic];
  }
  return;
}
```

Fonte: Produzida pelo autor

Figura 16 – Código em C++ da Implementação do acelerador em hardware para a base *Glass*.

```
#include "matmult.h"

// function to be accelerated in HW
template <typename T> void mmult_hw(T a[N][N], T b[N][N], T out[N][N]) {
float res[N][N];
L1:
  for (int ia = 0; ia < N; ++ia)
  L2:
    for (int ib = 0; ib < N; ++ib) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      res[ia][ib] = ((a[ia][ib] - b[ia][ib])*(a[ia][ib] - b[ia][ib]));
    }
int nv =9;   //tamanho do objeto da base Glass
int pop = 2; //número de candidatos por submatriz
int k = 6;   //número de grupos
int nvk=nv*k; //tamanho do candidato
float fitness[pop];
  for (int i =0; i < pop; ++i){
    float sum_t = 0;

    for (int ia = 0; ia < N; ++ia){
        float sum1 = 0;
        float sum2 = 0;
        float sum3 = 0;
        float sum4 = 0;
        float sum5 = 0;
        float sum6 = 0;
        int idx=i*nvk;
        for (int ib = 0; ib < nv; ++ib){
#pragma HLS pipeline
#pragma HLS loop_flatten off
            sum1 += res[ia][idx+ib];
            sum2 += res[ia][idx+ib+nv];
            sum3 += res[ia][idx+ib+nv*2];
            sum4 += res[ia][idx+ib+nv*3];
            sum5 += res[ia][idx+ib+nv*4];
            sum6 += res[ia][idx+ib+nv*5];
        }

        float a = sqrt(sum1);
        float b = sqrt(sum2);
        float c = sqrt(sum3);
        float d = sqrt(sum4);
        float e = sqrt(sum5);
        float f = sqrt(sum6);
        float smaller = min(a,b);
        smaller = min(smaller,c);
        smaller = min(smaller,d);
        smaller = min(smaller,e);
        smaller = min(smaller,f);
        sum_t += smaller;}
      fitness[i] = sum_t;
    }
  for (int ic = 0; ic < pop; ++ic){
      out[0][ic]=fitness[ic];
  }
  return;
}
```

Fonte: Produzida pelo autor

Figura 17 – Código em C++ da Implementação do acelerador em hardware para a base *CMC*.

```
#include "matmult.h"

// function to be accelerated in HW
template <typename T> void mmult_hw(T a[N][N], T b[N][N], T out[N][N]) {
float res[N][N];
L1:
  for (int ia = 0; ia < N; ++ia)
  L2:
    for (int ib = 0; ib < N; ++ib) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
      res[ia][ib] = ((a[ia][ib] - b[ia][ib])*(a[ia][ib] - b[ia][ib]));
    }
int nv =9;  //tamanho do objeto da base CMC
int pop = 4; //número de candidatos por submatriz
int k = 3;  //número de grupos
int nvk=nv*k; //tamanho do candidato
float fitness[pop];
  for (int i =0; i < pop; ++i){
    float sum_t = 0;

    for (int ia = 0; ia < N; ++ia){
        float sum1 = 0;
        float sum2 = 0;
        float sum3 = 0;
        int idx=i*nvk;
        for (int ib = 0; ib < nv; ++ib){
#pragma HLS pipeline
#pragma HLS loop_flatten off
            sum1 += res[ia][idx+ib];
            sum2 += res[ia][idx+ib+nv];
            sum3 += res[ia][idx+ib+nv*2];
        }

        float a = sqrt(sum1);
        float b = sqrt(sum2);
        float c = sqrt(sum3);
        float smaller = min(a,b);
        smaller = min(smaller,c);
        sum_t += smaller;}
      fitness[i] = sum_t;
    }
  for (int ic = 0; ic < pop; ++ic){
      out[0][ic]=fitness[ic];
  }
  return;
}
```

Fonte: Produzida pelo autor

APÊNDICE B

Código em Python: Montagem das submatrizes dos conjuntos de dados *Iris*, *Wine*, *Glass* e *CMC*

Todo código desenvolvido e apresentado neste Apêndice não foi otimizado. O objetivo é compartilhar os códigos que foram utilizados para extrair os resultados deste trabalho.

A Figura 18 apresenta 3 blocos de código. O bloco do topo representa o código para montagem das submatrizes da base *Iris*. O bloco do meio representa o código para montagem das submatrizes da base *Wine*. E o último bloco representa o código da montagem das submatrizes da base Glass.

Figura 18 – Código da Montagem das Bases *Iris*, *Wine* e *Glass*

.

In [ ]:
```python
import pandas as pd # biblioteca Python
import numpy as np # biblioteca Python
iris = pd.read_csv("./iris_csv.csv") # carrega a base Íris
iris = iris.drop(["class"], axis=1) # retira a coluna "class"
iris_1 = iris.drop(iris.index[120:150]) # estrutura os dados da submatriz
iris_array_1 = np.tile(np.array(iris_1), (1, 30)) # salva os dados na submatriz complementar
A1 = iris_array_1.astype(dtype=np.float32) # salva os dados da submatriz A1
iris_array_2 = np.tile([np.zeros(4)],(120,30)) # estrutura dos dados da submatriz complementar em dois passos. Passo 1
iris_array_2 [0:30][0:120] = np.tile(iris[120:150],30) # estrutura os dados na submatriz complementar. Passo 2
A2 = iris_array_2.astype(dtype=np.float32) # salva os dados na submatriz complementar A2
```

In [ ]:
```python
data = pd.read_csv("./Wine.csv") # carrega a base Wine
data = data.drop(["Wine"], axis=1) # retira a coluna "Wine"
data_1 = data.drop(data.index[120:178])  # estrutura os dados da submatriz em 3 passos. Passo 1
data_array_a_1 = np.tile(np.array(data_1), (1, 9)) # estrutura os dados da submatriz. Passo 2
data_array_a_1 = np.concatenate([data_array_a_1,np.zeros((120,3))],axis = 1) # estrutura os dados da submatriz. Passo 3
A1 = data_array_a_1.astype(dtype=np.float32) # salva os dados da submatriz A1
data_array_a_2 = np.tile([np.zeros(13)],(120,9)) # estrutura os dados da submatriz complementar em 3 passos. Passo 1
data_array_a_2 [0:58][0:120] = np.tile(data[120:178],9)  # estrutura os dados da submatriz complementar. Passo 2
data_array_a_2 = np.concatenate([data_array_a_2,np.zeros((120,3))],axis = 1)  # estrutura os dados da submatriz. Passo 3
A2 = data_array_a_2.astype(dtype=np.float32)# salva os dados da submatriz A2
```

In [ ]:
```python
data = pd.read_csv("./glass_csv.csv") # carrega a base Glass
data = data.drop(["Type"], axis=1) # retira a coluna "Type"
data_1 = data.drop(data.index[120:214]) # estrutura os dados da submatriz em 3 passos. Passo 1
data_array_a_1 = np.tile(np.array(data_1), (1,13)) # estrutura os dados da submatriz. Passo 2
data_array_a_1 = np.concatenate([data_array_a_1,np.zeros((120,3))],axis = 1) # estrutura os dados da submatriz. Passo 3
A1 = data_array_a_1.astype(dtype=np.float32) # salva os dados da submatriz A1
data_array_a_2 = np.tile([np.zeros(9)],(120,13)) # estrutura os dados da submatriz em 3 passos. Passo 1
data_array_a_2 [0:94][0:120] = np.tile(data[120:214],13) # estrutura os dados da submatriz. Passo 2
data_array_a_2 = np.concatenate([data_array_a_2,np.zeros((120,3))],axis = 1) # estrutura os dados da submatriz. Passo 3
A2 = data_array_a_2.astype(dtype=np.float32) # salva os dados da submatriz A2
```

Fonte: Produzida pelo autor

Figura 19 – Código da Montagem da Base *CMC*.

```
In [ ]: cmc = pd.read_csv("./cmc.data") # carrega a base Glass
cmc = cmc.drop(["Contraceptive_method_used"], axis=1) # retira a coluna "Contraceptive_method_used"
cmc_array_1 = np.tile(np.array(cmc[0:120][:]), (1,13)) # estrutura dos dados da submatriz complementar em dois passos. Passo 1
cmc_array_1 = np.concatenate([cmc_array_1,np.zeros((120,3))],axis = 1) # estrutura dos dados da submatriz complementar. Passo 2
A1 = cmc_array_1.astype(dtype=np.float32) # salva os dados da submatriz A1
cmc_array_2 = np.tile(np.array(cmc[120:240][:]), (1,13))# estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_2 = np.concatenate([cmc_array_2,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A2 = cmc_array_2.astype(dtype=np.float32)  # salva os dados da submatriz A2
cmc_array_3 = np.tile(np.array(cmc[240:360][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_3 = np.concatenate([cmc_array_3,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A3 = cmc_array_3.astype(dtype=np.float32)  # salva os dados da submatriz A3
cmc_array_4 = np.tile(np.array(cmc[360:480][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_4 = np.concatenate([cmc_array_4,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A4 = cmc_array_4.astype(dtype=np.float32)  # salva os dados da submatriz A4
cmc_array_5 = np.tile(np.array(cmc[480:600][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_5 = np.concatenate([cmc_array_5,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A5 = cmc_array_5.astype(dtype=np.float32) # salva os dados da submatriz A5
cmc_array_6 = np.tile(np.array(cmc[600:720][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_6 = np.concatenate([cmc_array_6,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A6 = cmc_array_6.astype(dtype=np.float32) # salva os dados da submatriz A6
cmc_array_7 = np.tile(np.array(cmc[720:840][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_7 = np.concatenate([cmc_array_7,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A7 = cmc_array_7.astype(dtype=np.float32) # salva os dados da submatriz A7
cmc_array_8 = np.tile(np.array(cmc[840:960][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_8 = np.concatenate([cmc_array_8,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A8 = cmc_array_8.astype(dtype=np.float32) # salva os dados da submatriz A8
cmc_array_9 = np.tile(np.array(cmc[960:1080][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_9 = np.concatenate([cmc_array_9,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A9 = cmc_array_9.astype(dtype=np.float32) # salva os dados da submatriz A9
cmc_array_10 = np.tile(np.array(cmc[1080:1200][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_10 = np.concatenate([cmc_array_10,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A10 = cmc_array_10.astype(dtype=np.float32) # salva os dados da submatriz A10
cmc_array_11 = np.tile(np.array(cmc[1200:1320][:]), (1,13))#estrutura dos dados da submatriz em dois passos. Passo 1
cmc_array_11 = np.concatenate([cmc_array_11,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A11 = cmc_array_11.astype(dtype=np.float32) # salva os dados da submatriz A11
cmc_array_12 = np.tile(np.array(cmc[1320:1440][:]), (1,13))#estrutura dos dados da submatrizem dois passos. Passo 1
cmc_array_12 = np.concatenate([cmc_array_12,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz. Passo 2
A12 = cmc_array_12.astype(dtype=np.float32) # salva os dados da submatriz A12
cmc_array_13 = np.tile(np.array(cmc[1440:1473][:]), (1,13))#estrutura dos dados da submatriz complementar em 3 passos.Passo 1
cmc_array_13 = np.concatenate([cmc_array_13,np.zeros((87,117))],axis = 0)# estrutura dos dados da submatriz complementar.Passo 2
cmc_array_13 = np.concatenate([cmc_array_13,np.zeros((120,3))],axis = 1)# estrutura dos dados da submatriz complementar.Passo 3
A13 = cmc_array_13.astype(dtype=np.float32) # salva os dados da submatriz A13
```

Fonte: Produzida pelo autor

APÊNDICE C

Código em Python: Montagem das submatrizes da população

Todo código desenvolvido e apresentado neste Apêndice não foi otimizado. O objetivo é compartilhar os códigos que foram utilizados para extrair os resultados deste trabalho.

As Figuras 20, 21, 22 e 23 mostram para cada conjunto de dados, 3 definições de funções em Python são implementadas. Uma função chamada de "*gera_candidato*" que gera um candidato conforme características do conjunto de dados, uma função chamada "*gera_pop*" que gera a quantidade de submatrizes para adequar a quantidade de candidatos à solução e uma função chamada de "*trata_dados*" que serve para atualizar as submatrizes complementares, a cada iteração do algoritmo BH.

Para os conjuntos *Wine* (Figura 21), *Glass* (Figura 22) e *CMC* (Figura 23) seguem as mesmas premissas, porém conforme as características de cada conjunto de dados, as soluções candidatas possuem tamanhos diferentes.

Figura 20 – Código em Python: Montagem das submatrizes da população para o conjunto *Iris*

.

```
#########################################################################
def gera_pop():  #definição da função que gera uma submatriz da população
    cand = (np.tile(np.array(gera_candidato()),(120,1)))  # gera 1 candidato
    table_pop=cand # salva o primerio candidato gerado
    for i in range (pop-1):  ## laço que controla o número de candidatos por submatriz (pop=número de candidatos por submatriz)

        cand = (np.tile(np.array(gera_candidato()),(120,1)))   # gera o próximo candidato
        table_pop = np.concatenate((table_pop,cand),axis = 1)  # salva o candidato gerado

    B = table_pop.astype(dtype=np.float32)  # salva em B a submatriz população
    B1= np.tile([np.zeros(12)],(120,10)) # prepara a configuração da submatriz complementar
    B1[0:30][0:120]=B[0:30][0:120] # monta a matriz complementar de B
    B1=B1.astype(dtype=np.float32) #salva em B1 a matriz complementar de B

    return B1,B

#########################################################################
def gera_candidato():  ##definição da função que gera uma solução candidata
    individuo = []
    for j in range (k): # (k = número de grupos da base de dados)

        for i in range(nv):  # nv = tamanho do objeto da base de dados
            individuo.append(np.random.uniform(bounds[i][0],bounds[i][1])) # gera candidato à solução
            # bounds[i][0] = limite máximo do espaço de busca e bounds[i][0] = limite mínimo do espaço de busca
    return individuo

#########################################################################
def trata_dados(Y):  # definição da função que atualiza as submatrizes complementares
    Y1= np.tile([np.zeros(12)],(120,10))  ## gera uma submatriz 120x120 (nula) para receber candidatos de tamanho 12.
    Y1[0:30][0:120]=Y[0:30][0:120]  # reserva espaço para receber os dados que devem ser considerados nos calculos matriciais
    Y1=Y1.astype(dtype=np.float32) # salva a submatriz complemento conforme dadps da submatriz de entrada Y

    return Y1
```

Fonte: Produzida pelo autor

Figura 21 – Código em Python: Montagem das submatrizes da população para o conjunto *Wine*

.

```
#########################################################################
def gera_pop():  #definição da função que gera uma submatriz da população
    cand = (np.tile(np.array(gera_candidato()),(120,1))) # gera 1 candidato
    table_pop=cand  # salva o primerio candidato gerado
    for i in range (pop-1): ## laço que controla o número de candidatos por submatriz (pop=número de candidatos por submatriz)

        cand = (np.tile(np.array(gera_candidato()),(120,1)))     # gera o próximo candidato
        table_pop = np.concatenate((table_pop,cand),axis = 1)    # salva o candidato gerado

    B1 = table_pop.astype(dtype=np.float32) # salva em B1 a submatriz população
    B1 = np.concatenate([B1,np.zeros((120,3))],axis = 1) # completa as ultimas 3 colunas com zero
    B1 = B1.astype(dtype=np.float32)   # salva em B1 a submatriz população
    B2 = np.zeros((120,120))  # prepara a configuração da submatriz complementar
    B2[0:58][0:120]=B1[0:58][0:120]  # monta a matriz complementar de B1
    B2=B2.astype(dtype=np.float32) #salva em B2 a matriz complementar de B1

    return B2,B1
#########################################################################
def gera_candidato():  ##definição da função que gera uma solução candidata
    individuo = []
    for j in range (k):  # (k = número de grupos da base de dados)

        for i in range(nv):  # nv = tamanho do objeto da base de dados
            individuo.append(np.random.uniform(bounds[i][0],bounds[i][1])) # gera candidato à solução
            # bounds[i][0] = limite máximo do espaço de busca e bounds[i][0] = limite mínimo do espaço de busca
    return individuo

#########################################################################
def trata_dados(Y):  # definição da função que atualiza as submatrizes complementares
    Y1 = np.zeros((120,120))  ## gera uma submatriz 120x120 (nula) para receber candidatos de tamanho 39.
    Y1[0:58][0:120]=Y[0:58][0:120]  # reserva espaço para receber os dados que devem ser considerados nos calculos matriciais
    Y1=Y1.astype(dtype=np.float32)  # salva a submatriz complemento conforme dadps da submatriz de entrada Y

    return Y1
```

Fonte: Produzida pelo autor

Figura 22 – Código em Python: Montagem das submatrizes da população para o conjunto *Glass*

.

```
#############################################################################
def gera_pop(): #definição da função que gera uma submatriz da população
    cand = (np.tile(np.array(gera_candidato()),(120,1))) # gera 1 candidato
    table_pop=cand   # salva o primerio candidato gerado
    for i in range (pop-1): # laço que controla o número de candidatos por submatriz (pop=número de candidatos por submatriz)

        cand = (np.tile(np.array(gera_candidato()),(120,1)))    # gera o próximo candidato
        table_pop = np.concatenate((table_pop,cand),axis = 1)   # salva o candidato gerado

    ###### como cada candidato tem tamanho 54. São considerados 2 candidatos por submatriz, total de 108.
    dime = (120,12) # monta matriz de tamanho 120x12.
    cp = np.zeros(dime)   # salva em cp matriz nula de tamanho 120x12.
    table_pop = np.concatenate((table_pop,cp),axis = 1)   # Este passo é para anular as 12 ultimas colunas
    B = table_pop.astype(dtype=np.float32) # salva em B a submatriz população
    B1= np.tile([np.zeros(30)],(120,4))  # prepara a configuração da submatriz complementar
    B1[0:94][0:120]=B[0:94][0:120]  # monta a matriz complementar de B
    B1=B1.astype(dtype=np.float32) #salva em B1 a matriz complementar de B

    return B1,B

#############################################################################
def gera_candidato():  ##definição da função que gera uma solução candidata
    individuo = []
    for j in range (k):  # (k = número de grupos da base de dados)

        for i in range(nv):    # nv = tamanho do objeto da base de dados
            individuo.append(np.random.uniform(bounds[i][0],bounds[i][1])) # gera candidato à solução
            # bounds[i][0] = limite máximo do espaço de busca e bounds[i][0] = limite mínimo do espaço de busca
    return individuo

#############################################################################
def trata_dados(Y):  # definição da função que atualiza as submatrizes complementares
    Y1= np.tile([np.zeros(12)],(120,10))  ## gera uma submatriz 120x120 (nula) para receber candidatos de tamanho 54.
    Y1[0:94][0:120]=Y[0:94][0:120]  # reserva espaço para receber os dados que devem ser considerados nos calculos matriciais
    Y1=Y1.astype(dtype=np.float32)  # salva a submatriz complemento conforme dados da submatriz de entrada Y

    return Y1
```

Fonte: Produzida pelo autor

Figura 23 – Código em Python: Montagem das submatrizes da população para o conjunto *CMC*

.

```
#############################################################################
def gera_pop():  #definição da função que gera uma submatriz da população
    cand = (np.tile(np.array(gera_candidato()),(120,1)))  # gera 1 candidato
    table_pop=cand  # salva o primerio candidato gerado
    for i in range (pop-1):  # laço que controla o número de candidatos por submatriz (pop=número de candidatos por submatriz)

        cand = (np.tile(np.array(gera_candidato()),(120,1)))     # gera o próximo candidato
        table_pop = np.concatenate((table_pop,cand),axis = 1)    # salva o candidato gerado

    ###### como cada candidato tem tamanho 27. São considerados 4 candidatos por submatriz, total de 108.
    dime = (120,12) # monta matriz de tamanho 120x12.
    cp = np.zeros(dime)    # salva em cp matriz nula de tamanho 120x12.
    table_pop = np.concatenate((table_pop,cp),axis = 1)  # Este passo é para anular as 12 ultimas colunas
    B = table_pop.astype(dtype=np.float32) # salva em B a submatriz população
    B1= np.tile([np.zeros(30)],(120,4)) # prepara a configuração da submatriz complementar
    B1[0:33][0:120]=B[0:33][0:120]  # monta a matriz complementar de B
    B1=B1.astype(dtype=np.float32)  #salva em B1 a matriz complementar de B

    return B1,B
#############################################################################
def gera_candidato():  ##definição da função que gera uma solução candidata
    individuo = []
    for j in range (k):  # (k = número de grupos da base de dados)

        for i in range(nv):  # nv = tamanho do objeto da base de dados
            individuo.append(np.random.uniform(bounds[i][0],bounds[i][1])) # gera candidato à solução
            # bounds[i][0] = limite máximo do espaço de busca e bounds[i][0] = limite mínimo do espaço de busca
    return individuo
#############################################################################
def trata_dados(Y):    # definição da função que atualiza as submatrizes complementares
    Y1= np.tile([np.zeros(12)],(120,10)) ## gera uma submatriz 120x120 (nula) para receber candidatos de tamanho 54.
    Y1[0:33][0:120]=Y[0:33][0:120]  # reserva espaço para receber os dados que devem ser considerados nos calculos matriciais
    Y1=Y1.astype(dtype=np.float32)   # salva a submatriz complemento conforme dadps da submatriz de entrada Y

    return Y1
```

Fonte: Produzida pelo autor

APÊNDICE D

Implementação do *Overlay* em Python

Todo código desenvolvido e apresentado neste Apêndice não foi otimizado. O objetivo é compartilhar os códigos que foram utilizados para extrair os resultados deste trabalho.

Para cada conjunto de dados, foi utilizado um *Overlay* dedicado. A Figura 24 mostra as bibliotecas Python necessárias para uso de *Overlays*, assim como a configuração inicial, e um função chamada "***run_kernel***" foi definida, indicando os parâmetros de controle de envio e recebimento de dados no buffer do acelerador. Dessa forma, uma função chamada "***accel_hw***" foi desenvolvida e definida para convocação do acelerador em hardware ao longo do algoritmo BH implementado. Conforme explicado no texto principal, o acelerador computa 2 matrizes de tamanho 120x120 por vez, e portanto, uma estrutura de repetição utilizando o acelerador foi necessário, uma vez que os conjuntos de dados são fornecidos por mais de uma submatriz. Como os conjuntos *Iris*, *Wine* e *Glass* são representados por duas submatrizes, o mesmo código pode ser representado na Figura 25. O código desenvolvido para utilizar o acelerador para o conjunto *CMC* possui uma estrutura de repetição compatível com a quantidade de submatrizes que representa todo o conjunto (13 submatrizes). Ao final de cada função ***accel_hw***, obtem-se a função objetiva das soluções candidatas representas na submatriz população de entrada. Esta função é bem utilizada ao longo a implementação BH, uma vez que os experimentos deste estudo, utilizou quantidades de soluções candidatas que foram representadas quantidade de submatrizes mostradas na Tabela 2.

Figura 24 – Código em Python: Configuração do *Overlay* para as Bases *Iris*, *Wine*, *Glass* e *CMC*

```
from pynq import (allocate, Overlay) # biblioteca Python
import numpy as np # biblioteca Python
################################################################################
overlay = Overlay('./matmult.bit') # arquivo (acelerador HW) gerado no Vivado HLS e Vivado Design Suite
dma = overlay.dma # carrega overlay dma (configuração fornecida na Pynq-Z1)
mmult_ip = overlay.accel # carrega overlay gerado para o acelerador em HW
################################################################################
DIM = 120 #define o tamanho da matrix
in_buffer = allocate(shape=(2, DIM, DIM), dtype=np.float32, cacheable=False) #configuração do buffer de entrada
out_buffer = allocate(shape=(DIM, DIM), dtype=np.float32, cacheable=False) #configuração do buffer de saída
################################################################################
CTRL_REG = 0x00 #configuração do registrador do buffer
AP_START = (1<<0) #configuração do registrador do buffer
AUTO_RESTART = (1<<7)  #configuração do registrador do buffer

def run_kernel(): # definição da função do acelerador em HW
    dma.sendchannel.transfer(in_buffer) # fluxo de dados entre o dma e o buffer de entrada
    dma.recvchannel.transfer(out_buffer) # fluxo de dados entre o buffer de saída e o dma
    mmult_ip.write(CTRL_REG, (AP_START | AUTO_RESTART))  # atualiza os dados nos registradores
    dma.sendchannel.wait() # comando de sincronismo do canal para liberação do buffer de entrada receber novos dados
    dma.recvchannel.wait()  # comando de sincronismo do canal para liberação do buffer de saída
```

Fonte: Fonte adaptada da referência (BAGNI et al., 2016)

Figura 25 – Código da função que executa o acelerador para os conjuntos *Iris*, *Wine* e *Glass*

.

```
############## For each star, evaluate the objective function ###############
def accel_hw(x1,x): # definição da função do acelerador em HW. Nota: Válida para as bases Iris, Wine e Glass
# variavel x é um submatriz da população e x1 é a submatriz complemento da população.
    in_buffer[:] = np.stack((A1, x)) # carrega o buffer do acelerador com a submatriz A1 (base) e submatriz x (população)
    run_kernel() # executa o acelerador em HW
    candidato=[] # inicia lista que salva os valores de fo das estrelas da submatriz x
    for i in range (pop): # inicia laço para salvar os dados de saída do acelerador
        candidato.append(out_buffer[0][i]) # lista salva os valores de saída do acelerador em hw

    in_buffer[:] = np.stack((A2, x1)) # carrega o buffer com a submatriz complementar A2 (base) e x1 (população)
    run_kernel()   # executa o acelerador em HW

    for i in range (pop): # inicia laço para salvar os dados de saída do acelerador
        candidato[i]=candidato[i]+(out_buffer[0][i])  # lista salva os valores de saída do acelerador em hw

    return candidato# retorna os valores de fo das estrelas contidas nas submatrizes A1 e A2
```

Fonte: Produzida pelo autor

Figura 26 – Código da função que executa o acelerador para o conjunto *CMC*

.

```
############## For each star, evaluate the objective function ###############
def accel_hw(x1,x): # definição da função do acelerador em HW. Nota: Válida para base CMC.
# variavel x é um submatriz da população e x1 é a submatriz complemento da população.
    in_buffer[:] = np.stack((A1, x)) # carrega o buffer do acelerador com a submatriz A1 (base) e submatriz x (população)
    run_kernel() # executa o acelerador em HW
    candidato=[]  # inicia lista que salva os valores de fo das estrelas da submatriz x
    for i in range (pop): # inicia laço para salvar os dados de saída do acelerador
        candidato.append(out_buffer[0][i]) # lista salva os valores de saída do acelerador em hw
    in_buffer[:] = np.stack((A2, x))  # carrega o buffer com a submatriz complementar A2 (base) e x (população)
    run_kernel()    # executa o acelerador em HW
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])
    in_buffer[:] = np.stack((A3, x)) # carrega o buffer com a submatriz complementar A3 (base) e x (população)
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])
    in_buffer[:] = np.stack((A4, x)) # carrega o buffer com a submatriz complementar A4 (base) e x (população)
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i]# carrega o buffer com a submatriz complementar A5 (base) e x (população)
    in_buffer[:] = np.stack((A5, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A6 (base) e x (população)
    in_buffer[:] = np.stack((A6, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A7 (base) e x (população)
    in_buffer[:] = np.stack((A7, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i]) # carrega o buffer com a submatriz complementar A8 (base) e x (população)
    in_buffer[:] = np.stack((A8, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A9 (base) e x (população)
    in_buffer[:] = np.stack((A9, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A10 (base) e x (população)
    in_buffer[:] = np.stack((A10, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A11 (base) e x (população)
    in_buffer[:] = np.stack((A11, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A12 (base) e x (população)
    in_buffer[:] = np.stack((A12, x))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])# carrega o buffer com a submatriz complementar A13 (base) e x1 (população)
    in_buffer[:] = np.stack((A13, x1))
    run_kernel()
    for i in range (pop):
        candidato[i]=candidato[i]+(out_buffer[0][i])
    return candidato
```

Fonte: Produzida pelo autor

APÊNDICE E

Código em Python: Implementação do algoritmo *Black Hole*

Todo código desenvolvido e apresentado neste Apêndice não foi otimizado. O objetivo é compartilhar os códigos que foram utilizados para extrair os resultados deste trabalho. A solução proposta foi implementada em Python e pode ser definida seguindo os passos:

- ❏ Definição dos parâmetros conforme conjunto de dados (Veja Figura 27);
- ❏ Carrega o conjunto de dados e monta as submatrizes, conforme mostrado no Apêndice B;
- ❏ Definição dos parâmetros iniciais: número máximo de iterações "(*MaxIter*)" e tamanho da população "(*parametro*)" , seguindo a Tabela 2;
- ❏ Montagem das submatrizes da população conforme Apêndice C;
- ❏ Definições e configurações da *Overlay*, conforme mostrado no Apêndice D;
- ❏ Implementação do código principal: Para *Iris* segue o código na Figura 28;
- ❏ Implementação do código principal: Para *Wine* segue o código na Figura 29;
- ❏ Implementação do código principal: Para *Glass* segue o código na Figura 30;
- ❏ Implementação do código principal: Para *CMC* segue o código na Figura 31.

Figura 27 – Definição dos parâmetros dos conjuntos de dados: *Iris*, *Wine*, *Glass* e *CMC*. O tamanho da solução candidata e limitações máximas e mínimas do espaço de busca também são definidos neste bloco de código.

```python
import time # biblioteca Python
import matplotlib.pyplot as plt  # biblioteca Python
from scipy.spatial import distance  # biblioteca Python
import math # biblioteca Python
def selecao(bh_index):  # definição de função para representar o BH de forma matricial
    x=bh_index%pop   # indice que localiza a solução candidata dentro das submatrizes da população
    y=math.trunc(bh_index/pop)   # indice que localiza a solução candidata dentro da submatriz da população
    bh = Vet[y][0][x*nvk:x*nvk+nvk:]  # montagem do Bh

    return bh.astype(dtype=np.float32)
```

```python
k = 3 # número de grupos do conjunto Iris
nv = 4  # tamanho do objeto Iris
pop = 10 # size of population
nvk=nv*k
bounds=[] # setup list of max and min (irisdf)

for i in range (nv):
    bounds.append((iris.max().get(i),iris.min().get(i))) # upper and lower bounds of variables
```

```python
k = 3 # número de grupos do conjunto Wine
nv = 13 # tamanho do objeto Wine
pop = 3 # quantidade de soluçõeos candidatas por submatriz
nvk=nv*k  # tamanho da solução candidata
bounds=[] # vetor que armazenará os limites do espaço de busca

for i in range (nv): # define os limites máximos e mínimos do espaço de busca
    bounds.append((data.max().get(i),data.min().get(i)))   # data = conjunto de dados
```

```python
k = 3 # # número de grupos do conjunto CMC
nv = 9 # tamanho do objeto CMC
pop = 4 # quantidade de soluçõeos candidatas por submatriz
nvk=nv*k  # quantidade de soluçõeos candidatas por submatriz
bounds=[] ## vetor que armazenará os limites do espaço de busca

for i in range (nv): # define os limites máximos e mínimos do espaço de busca
    bounds.append((cmc.max().get(i),cmc.min().get(i))) # data = conjunto de dados
```

```python
k = 6 # número de grupos do conjunto Glass
nv = 9 # tamanho do objeto Glass
pop = 2 # quantidade de soluçõeos candidatas por submatriz
nvk=nv*k  # quantidade de soluçõeos candidatas por submatriz
bounds=[] # vetor que armazenará os limites do espaço de busca

for i in range (nv): # define os limites máximos e mínimos do espaço de busca
    bounds.append((data.max().get(i),data.min().get(i))) # data = conjunto de dados
```

Fonte: Produzida pelo autor

Figura 28 – Implementação do algoritmo BH para o conjunto *Iris*

```
P_global =[] # Armazena os dados dos melhores resultados obtido a cada simulação
best_fitness_global=float("inf") # Armazena melhores resultados obtido a cada simulação
P_time =[] # Armazena os dados de tempo de execução
Runs=50 # define o número de simulações
MaxIter=10000 # define o número de iterações por simulação
parametro = 20 # parâmetro de controle de quantidade de matrizes particionadas para representar a população
for Simulations in range (Runs):  # Inicio do laço de simulações
    best_fitness_local=float("inf") # inicializa o parâmetro para um problema de minimização
    Vet=[] # lista para armazenar submatrizes
    Vet1=[] # lista para armazenar submatrizes complementares de Vet
    for i in range (parametro): # Inicio do laço de uma população inteira (número de estrelas = pop)
        (Mat1,Mat)=gera_pop() #gera submatriz B e complemento de B
        Vet.append(Mat) # armazena submatriz B
        Vet1.append(Mat1) # armazena submatriz complementar de B
    start_time=time.time() # inicio do contador de tempo de execução
    for Loop in range (MaxIter): # Inicio do laço da simulação (Simulation)
        candidato = accel_hw(Vet1[0],Vet[0]) # calcula função objetiva (fo) das estrelas contidas na primeira submatriz B
        pool = candidato # armazena os valores da fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1): # Inicio do laço que continua o calculo da fo das próximas estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato)) # armazena os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool): # verifica qual estrela será o BH
            best_fitness_local=min(pool) # armazena o melhor fo
            bh_index=np.argmin(pool) # salva qual estrela será o BH.
            bh=selecao(bh_index) # monta submatriz para representar o BH, em dois passos. Passo 1
            bh1=np.tile(np.array(bh),(120,10)) # monta submatriz para representar o BH. Passo 2
        for i in range (parametro): # Início do laço que movimenta as estrelas no espaço de busca em direção ao BH
            rand=np.random.random() # gera um número aleatório entre 0 e 1 utilizando a biblioteca numpy
            Vet[i]=rand*(bh1-Vet[i])+Vet[i] # Eq. 20
            Vet1[i]=trata_dados(Vet[i]) # atualiza a submatriz B, após Eq. 20
        candidato = accel_hw(Vet1[0],Vet[0]) # calcula fo das estrelas contidas na primeira submatriz B atualizada
        pool = candidato # atualiza os valores de fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1): # Início do laço que continua o calculo de fo das estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato)) # atualiza os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool): # verifica qual estrela será o BH
            best_fitness_local=min(pool) # armazena o melhor fo
            bh_index=np.argmin(pool)   # salva qual estrela será o BH.
            bh=selecao(bh_index) # monta submatriz para representar o BH, em dois passos. Passo 1
            bh1=np.tile(np.array(bh),(120,10))   # monta submatriz para representar o BH. Passo 2
        raio=(best_fitness_local/np.sum(pool) ) # calcula o raio de ação do horizonte de eventos do BH (Eq. 19)
        for i in range (parametro): # Início do laço que verifica se alguma estrela será sugada pelo BH
            for j in range (pop): # Início do laço de pop
                bh2=bh.reshape(1,nvk) # estrutura os dados que representam o BH em um vetor
                star=Vet[i][0][j*nvk:j*nvk+nvk:].reshape(1,nvk) # localiza a estrela corrente dentro da submatriz B
                distancia=distance.cdist(bh2,star) # calcula a distancia entre o BH e a estrela corrente
                if distancia<=raio: # valida se a estrela corrente será sugada
                    Vet[i][0][j*nvk:j*nvk+nvk:]=np.array(gera_candidato()) # Se sim, uma nova estrela é gerada
                    xx=Vet[i][0][j*nvk:j*nvk+nvk:] # estrutura dados para atualizar a submatriz B com a nova estrela
                    for f in range (119): # Início do laço que percorre uma submatriz
                        Vet[i][f+1][j*nvk:j*nvk+nvk:]=xx # indexa a posição da nova estrela no lugar da estrela morta
                    Vet1[i]=trata_dados(Vet[i]) # atualiza a submatriz B com a nova estrela gerada
        if best_fitness_local<best_fitness_global: # Checa qual o melhor resultado até o momento
            best_fitness_global=best_fitness_local # armazena o melhor valor de fo até o momento
            best_bh=bh # armazena o melhor resultado de fo
        P_global.append(best_fitness_global)        #### armazena o melhor resultado global
# Se o qdt de iterações (Loop) não chegou ao fim (MaxIter), inicia nova iteração, voltando para "for Loop in range (MaxIter):"
    stop_time=time.time() # fim da contagem de tempo de execução da simulação(Simulations)
    Time_CPU = stop_time-start_time # cálculo do tempo de execução da simulação(Simulations)
    P_time.append(Time_CPU) # salva o tempo de execução da simulação(Simulations)
# Se Simulations não chegou ao fim (Runs), inicia nova simulação, voltando para "for Simulations in range (Runs):"
######## Fim do algoritmo ###################################
```

Fonte: Produzida pelo autor

Figura 29 – Implementação do algoritmo BH para o conjunto *Wine*

```
P_global =[]  # Armazena os dados dos melhores resultados obtido a cada simulação
best_fitness_global=float("inf") # Armazena melhores resultados obtido a cada simulação
P_time =[]  # Armazena os dados de tempo de execução
Runs=50      # define o número de simulações
MaxIter=7000  # define o número de iterações por simulação
parametro = 67  # parâmetro de controle de quantidade de matrizes particionadas para representar a população
for Simulations in range (Runs):    # Inicio do laço de simulações
    best_fitness_local=float("inf") # inicializa o parâmetro para um problema de minimização
    Vet=[]   # lista para armazenar submatrizes
    Vet1=[]  # lista para armazenar submatrizes complementares de Vet
    for i in range (parametro): # Inicio do laço de uma população inteira (número de estrelas = pop)
        (Mat1,Mat)=gera_pop()  #gera submatriz B e complemento de B
        Vet.append(Mat)  # armazena submatriz B
        Vet1.append(Mat1)    # armazena submatriz complementar de B
    start_time=time.time()   # inicio do contador de tempo de execução
    for Loop in range (MaxIter):  # Início do laço da simulação (Simulation)
        candidato = accel_hw(Vet1[0],Vet[0]) # calcula função objetiva (fo) das estrelas contidas na primeira submatriz B
        pool = candidato  # armazena os valores da fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Inicio do laço que continua o calculo da fo das próximas estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1])  # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato)) # armazena os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool): # verifica qual estrela será o BH
            best_fitness_local=min(pool) # armazena o melhor fo
            bh_index=np.argmin(pool)  # salva qual estrela será o BH.
            bh=selecao(bh_index) # monta submatriz para representar o BH, em 3 passos. Passo 1
            bh1=np.tile(np.array(bh),(120,3)) # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,3))],axis = 1) # monta submatriz para representar o BH. Passo 3
        for i in range (parametro): # Início do laço que movimenta as estrelas no espaço de busca em direção ao BH
            rand=np.random.random() # gera um número aleatório entre 0 e 1 utilizando a biblioteca numpy
            Vet[i]=rand*(bh1-Vet[i])+Vet[i] # Eq. 20
            Vet1[i]=trata_dados(Vet[i])         # atualiza a submatriz B, após Eq. 20
        candidato = accel_hw(Vet1[0],Vet[0]) # calcula fo das estrelas contidas na primeira submatriz B atualizada
        pool = candidato # atualiza os valores de fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Início do laço que continua o calculo de fo das estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato)) # atualiza os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool): # verifica qual estrela será o BH
            best_fitness_local=min(pool) # armazena o melhor fo
            bh_index=np.argmin(pool)   # salva qual estrela será o BH.
            bh=selecao(bh_index)       # monta submatriz para representar o BH, em dois passos. Passo 1
            bh1=np.tile(np.array(bh),(120,3))  # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,3))],axis = 1) # calcula o raio de ação do horizonte de eventos do BH (Eq. 19)
        raio=(best_fitness_local/np.sum(pool) ) # Início do laço que verifica se alguma estrela será sugada pelo BH
        for i in range (parametro): # Início do laço que verifica se alguma estrela será sugada pelo BH
            for j in range (pop): # Início do laço de pop
                bh2=bh.reshape(1,nvk) # estrutura os dados que representam o BH em um vetor
                star=Vet[i][0][j*nvk:j*nvk+nvk:].reshape(1,nvk) # localiza a estrela corrente dentro da submatriz B
                distancia=distance.cdist(bh2,star)   # calcula a distancia entre o BH e a estrela corrente
                if distancia<=raio: # valida se a estrela corrente será sugada
                    Vet[i][0][j*nvk:j*nvk+nvk:]=np.array(gera_candidato()) # Se sim, uma nova estrela é gerada
                    xx=Vet[i][0][j*nvk:j*nvk+nvk:]  # estrutura dados para atualizar a submatriz B com a nova estrela
                    for f in range (119): # Início do laço que percorre uma submatriz
                        Vet[i][f+1][j*nvk:j*nvk+nvk:]=xx # indexa a posição da nova estrela no lugar da estrela morta
                    Vet1[i]=trata_dados(Vet[i])    # atualiza a submatriz B com a nova estrela gerada
        if best_fitness_local<best_fitness_global:  # Checa qual o melhor resultado até o momento
            best_fitness_global=best_fitness_local # armazena o melhor valor de fo até o momento
            best_bh=bh # armazena o melhor resultado de fo
        P_global.append(best_fitness_global)        #### armazena o melhor resultado global
# Se o qdt de iterações (Loop) não chegou ao fim (MaxIter), inicia nova iteração, voltando para "for Loop in range (MaxIter):"
    stop_time=time.time() # fim da contagem de tempo de execução da simulação(Simulations)
    Time_CPU = stop_time-start_time # cálculo do tempo de execução da simulação(Simulations)
    P_time.append(Time_CPU) # salva o tempo de execução da simulação(Simulations)
# Se Simulations não chegou ao fim (Runs), inicia nova simulação, voltando para "for Simulations in range (Runs):"
######## Fim do algoritmo ##########################################
```

Fonte: Produzida pelo autor

Figura 30 – Implementação do algoritmo BH para o conjunto *Glass*

```
P_global =[] # Armazena os dados dos melhores resultados obtido a cada simulação
P_time = [] # Armazena melhores resultados obtido a cada simulação
best_fitness_global=float("inf") # Armazena os dados de tempo de execução
Runs=50  # define o número de simulações
MaxIter=20000  # define o número de iterações por simulação
parametro=100  # parâmetro de controle de quantidade de matrizes particionadas para representar a população
for Simulations in range (Runs):     # Inicio do laço de simulações
    best_fitness_local=float("inf")  # inicializa o parâmetro para um problema de minimização
    Vet=[]  # lista para armazenar submatrizes
    Vet1=[] # lista para armazenar submatrizes complementares de Vet
    for i in range (parametro): # Inicio do laço de uma população inteira (número de estrelas = pop)
        (Mat1,Mat)=gera_pop()   #gera submatriz B e complemento de B
        Vet.append(Mat)         # armazena submatriz B
        Vet1.append(Mat1)      # armazena submatriz complementar de B
    start_time=time.time()     # inicio do contador de tempo de execução
    for Loop in range (MaxIter):  # Inicio do laço da simulação (Simulation)
        candidato = accel_hw(Vet1[0],Vet[0])  # calcula função objetiva (fo) das estrelas contidas na primeira submatriz B
        pool = candidato  # armazena os valores da fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Inicio do laço que continua o calculo da fo das próximas estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato)) # armazena os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool):  # verifica qual estrela será o BH
            best_fitness_local=min(pool)  # armazena o melhor fo
            bh_index=np.argmin(pool)      # salva qual estrela será o BH.
            bh=selecao(bh_index) # monta submatriz para representar o BH, em 3 passos. Passo 1
            bh1=np.tile(np.array(bh),(120,2)) # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,12))],axis = 1) # monta submatriz para representar o BH. Passo 3
        for i in range (parametro): # Início do laço que movimenta as estrelas no espaço de busca em direção ao BH
            rand=np.random.random() # gera um número aleatório entre 0 e 1 utilizando a biblioteca numpy
            Vet[i]=rand*(bh1-Vet[i])+Vet[i]  # Eq. 20
            Vet1[i]=trata_dados(Vet[i])            # atualiza a submatriz B, após Eq. 20
        candidato = accel_hw(Vet1[0],Vet[0])  # calcula fo das estrelas contidas na primeira submatriz B atualizada
        pool = candidato    # atualiza os valores de fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Início do laço que continua o calculo de fo das estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato))  # atualiza os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool):  # verifica qual estrela será o BH
            best_fitness_local=min(pool)  # armazena o melhor fo
            bh_index=np.argmin(pool)  # salva qual estrela será o BH.
            bh=selecao(bh_index)    # monta submatriz para representar o BH, em 3 passos. Passo 1
            bh1=np.tile(np.array(bh),(120,2))  # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,12))],axis = 1)  # monta submatriz para representar o BH. Passo 3
        raio=(best_fitness_local/np.sum(pool) )  # calcula o raio de ação do horizonte de eventos do BH (Eq. 19)
        for i in range (parametro):  # inicio do laço que verifica se alguma estrela mserá sugada pelo BH
            for j in range (pop):  # Início do laço de pop
                bh2=bh.reshape(1,nvk)  # estrutura os dados que representam o BH em um vetor
                star=Vet[i][0][j*nvk:j*nvk+nvk:].reshape(1,nvk) # localiza a estrela corrente dentro da submatriz B
                distancia=distance.cdist(bh2,star) # calcula a distancia entre o BH e a estrela corrente
                if distancia<=raio:  # valida se a estrela corrente será sugada
                    Vet[i][0][j*nvk:j*nvk+nvk:]=np.array(gera_candidato()) # Se sim, uma nova estrela é gerada
                    xx=Vet[i][0][j*nvk:j*nvk+nvk:] # estrutura dados para atualizar a submatriz B com a nova estrela
                    for fdp in range (119):  # Início do laço que percorre uma submatriz
                        Vet[i][fdp+1][j*nvk:j*nvk+nvk:]=xx  # indexa a posição da nova estrela no lugar da estrela morta
                    Vet1[i]=trata_dados(Vet[i])          # atualiza a submatriz B com a nova estrela gerada
        if best_fitness_local<best_fitness_global:     # Checa qual o melhor resultado até o momento
            best_fitness_global=best_fitness_local   # armazena o melhor valor de fo até o momento
            best_bh=bh # armazena o melhor resultado de fo
        P_global.append(best_fitness_global)        #### armazena o melhor resultado global
# Se o qdt de iterações (Loop) não chegou ao fim (MaxIter), inicia nova iteração, voltando para "for Loop in range (MaxIter):"
    stop_time=time.time()  # fim da contagem de tempo de execução da simulação(Simulations)
    Time_CPU = stop_time-start_time # cálculo do tempo de execução da simulação(Simulations)
    P_time.append(Time_CPU)  # salva o tempo de execução da simulação(Simulations)
        # Se Simulations não chegou ao fim (Runs), inicia nova simulação, voltando para "for Simulations in range (Runs):"
######## Fim do algoritmo ########################################
```

Fonte: Produzida pelo autor

Figura 31 – Implementação do algoritmo BH para o conjunto *CMC*

```
P_global =[] # Armazena os dados dos melhores resultados obtido a cada simulação
P_time = []  # Armazena melhores resultados obtido a cada simulação
best_fitness_global=float("inf")  # Armazena os dados de tempo de execução
Runs=50 # define o número de simulações
MaxIter=20000 # define o número de iterações por simulação
parametro=50  # parâmetro de controle de quantidade de matrizes particionadas para representar a população
for Simulations in range (Runs):      # Inicio do laço de simulações
    best_fitness_local=float("inf")   # inicializa o parâmetro para um problema de minimização
    Vet=[] # lista para armazenar submatrizes
    Vet1=[] # lista para armazenar submatrizes complementares de Vet
    for i in range (parametro):  # Inicio do laço de uma população inteira (número de estrelas = pop)
        (Mat1,Mat)=gera_pop()    #gera submatriz B e complemento de B
        Vet.append(Mat)          # armazena submatriz B
        Vet1.append(Mat1)        # armazena submatriz complementar de B
    start_time=time.time()      # inicio do contador de tempo de execução
    for Loop in range (MaxIter):  # Inicio do laço da simulação (Simulation)
        candidato = accel_hw(Vet1[0],Vet[0])  # calcula função objetiva (fo) das estrelas contidas na primeira submatriz B
        pool = candidato  # armazena os valores da fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Inicio do laço que continua o calculo da fo das próximas estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1])  # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato))   # armazena os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool):   # verifica qual estrela será o BH
            best_fitness_local=min(pool)   # armazena o melhor fo
            bh_index=np.argmin(pool)       # salva qual estrela será o BH.
            bh=selecao(bh_index)           # monta submatriz para representar o BH, em 3 passos. Passo 1
            bh1=np.tile(np.array(bh),(120,4)) # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,12))],axis = 1) # monta submatriz para representar o BH. Passo 3
        for i in range (parametro): # Início do laço que movimenta as estrelas no espaço de busca em direção ao BH
            rand=np.random.random() # gera um número aleatório entre 0 e 1 utilizando a biblioteca numpy
            Vet[i]=rand*(bh1-Vet[i])+Vet[i] # Eq. 20
            Vet1[i]=trata_dados(Vet[i])              # atualiza a submatriz B, após Eq. 20
        candidato = accel_hw(Vet1[0],Vet[0])  # calcula fo das estrelas contidas na primeira submatriz B atualizada
        pool = candidato    # atualiza os valores de fo das estrelas contidas na primeira submatriz B
        for i in range (parametro-1):  # Início do laço que continua o calculo de fo das estrelas até o fim de pop
            candidato = accel_hw(Vet1[i+1],Vet[i+1]) # calcula fo das estrelas contidas na submatriz B subsequente
            pool = np.concatenate((pool,candidato))  # atualiza os valores da fo das estrelas obtidas no passo anterior
        if best_fitness_local>min(pool):  # verifica qual estrela será o BH
            best_fitness_local=min(pool)  # armazena o melhor fo
            bh_index=np.argmin(pool)      # salva qual estrela será o BH.
            bh=selecao(bh_index)        # monta submatriz para representar o BH, em 3 passos. Passo 1
            bh1=np.tile(np.array(bh),(120,4))  # monta submatriz para representar o BH. Passo 2
            bh1=np.concatenate([bh1,np.zeros((120,12))],axis = 1)# monta submatriz para representar o BH. Passo 3
        raio=(best_fitness_local/np.sum(pool) ) # calcula o raio de ação do horizonte de eventos do BH (Eq. 19)
        for i in range (parametro): # inicio do laço que verifica se alguma estrela mserá sugada pelo BH
            for j in range (pop): # Início do laço de pop
                bh2=bh.reshape(1,nvk)    # estrutura os dados que representam o BH em um vetor
                star=Vet[i][0][j*nvk:j*nvk+nvk:].reshape(1,nvk) # localiza a estrela corrente dentro da submatriz B
                distancia=distance.cdist(bh2,star)       # calcula a distancia entre o BH e a estrela corrente
                if distancia<=raio:  # valida se a estrela corrente será sugada
                    Vet[i][0][j*nvk:j*nvk+nvk:]=np.array(gera_candidato()) # Se sim, uma nova estrela é gerada
                    xx=Vet[i][0][j*nvk:j*nvk+nvk:] # estrutura dados para atualizar a submatriz B com a nova estrela
                    for f in range (119):  # Início do laço que percorre uma submatriz
                        Vet[i][f+1][j*nvk:j*nvk+nvk:]=xx # indexa a posição da nova estrela no lugar da estrela morta
                    Vet1[i]=trata_dados(Vet[i])         # atualiza a submatriz B com a nova estrela gerada
        if best_fitness_local<best_fitness_global: # Checa qual o melhor resultado até o momento
            best_fitness_global=best_fitness_local # armazena o melhor valor de fo até o momento
            best_bh=bh # armazena o melhor resultado de fo
        P_global.append(best_fitness_global)       #### armazena o melhor resultado global
# Se o qdt de iterações (Loop) não chegou ao fim (MaxIter), inicia nova iteração, voltando para "for Loop in range (MaxIter):"
    stop_time=time.time() # fim da contagem de tempo de execução da simulação(Simulations)
    Time_CPU = stop_time-start_time  # cálculo do tempo de execução da simulação(Simulations)
    P_time.append(Time_CPU)  # salva o tempo de execução da simulação(Simulations)
    # Se Simulations não chegou ao fim (Runs), inicia nova simulação, voltando para "for Simulations in range (Runs):"
######## Fim do algoritmo ########################################
```

Fonte: Produzida pelo autor