UNIVERSIDADE FEDERAL RURAL DO SEMIÁRIDO - UFERSA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
Trabalho de Conclusão de Curso (2023).

PROJETO DE UM *HARDWARE* ACELERADOR DO ALGORITMO DE DISTÂNCIA EUCLIDIANA

Giovanni Moreira Guimarães[1], Sílvio R. Fernandes de Araújo[2]

[1]Graduando – webminst@hotmail.com
[2]Orientador – silvio@ufersa.edu.br

Resumo. Gigantescos volumes de dados são gerados diariamente tornando cada vez mais custoso para os computadores processá-los em tempo hábil, razão pela qual há uma demanda por soluções que possam acelerar os processos envolvidos. Este trabalho visa apresentar um desenho da arquitetura de um acelerador do cálculo da distância euclidiana em hardware avaliando as oportunidades de paralelismo e pipeline e uma análise da aceleração com base nos resultados de testes simulados e cálculos matemáticos com a finalidade de melhorar o tempo de resposta do algoritmo K-means. A pesquisa é baseada em um Programa Institucional de Bolsas de Iniciação Científica (PIBIC) realizado nos últimos 12 meses, que contou com revisão de literatura sobre o assunto e do uso do Logisim para a confecção do desenho e das simulações de testes e comparativos com versões em software mostrando que a aceleração aumenta com o volume de dados até se estabilizar próximo a 9 vezes mais eficiente quando comparado à versão puramente sequencial e atingindo uma melhoria de 8,5 em comparação com a implementação em assembly do processador MIPS. A partir dos resultados podemos concluir que o grande ganho de desempenho do acelerador está no uso de paralelismo na comparação de indivíduos com centroide e uso de *pipeline* nas etapas do cálculo da distância, além do paralelismo empregado na árvore de soma e de comparação usada na implementação.
Palavras-chave: FPGA; distância euclidiana; *K-means;* aceleração; *hardware.*

1. INTRODUÇÃO

A busca por melhor desempenho em processadores resultou em diversas técnicas e arquiteturas, como pipeline, processadores superescalares e o aumento da frequência do *clock*. No entanto, foi devido a preocupações com consumo de potência e dissipação de calor que surgiram os processadores multicores, que aumentam o desempenho por meio de execução paralela sem alterar o *clock*. Embora esses processadores sejam muito flexíveis, existem soluções mais eficientes em termos de tempo de computação, entre elas encontra-se o FPGA (*Field Programmable Gate Array*), que possui arquitetura reconfigurável, capaz de definir qualquer função digital em tempo de projeto, com alto potencial de paralelismo e baixo consumo de energia.

Estudos [1-3] indicam que a combinação de CPU-FPGA tem se mostrado uma solução promissora em termos de eficiência energética, devido ao alto desempenho, baixo consumo de energia e capacidade de reconfiguração para acelerar diferentes aplicações, em áreas como sequenciamento de DNA, ranqueamento de páginas na *Web*, redes neurais, processamento de imagens em tempo real e algoritmos de aprendizado de máquina. Dentre estas, especialmente o aprendizado de máquina, tem se tornado cada vez mais presente em várias áreas, como IoT (*Internet of Things*), Medicina, carros autônomos, processamento de linguagem natural, previsão de fenômenos etc. e, um dos principais algoritmos, o *K-means*, que agrupa os dados procurando padrões/semelhanças, tem sido amplamente utilizado.

O ganho de eficiência obtido com o FPGA tem tornado essa plataforma mais atrativa para desenvolvimento, uma vez que as perdas impostas por limitações de *hardware* podem ser em grande parte eliminadas adotando uma utilização mais eficiente dos recursos fornecidos, numa batalha constante entre performance e eficiência energética [4]. A aceleração de *hardware* fornecida pelo FPGA também tem sido aplicada com sucesso na aceleração de algoritmos de *Deep Learning*, em particular *Convolutional Neural Network* (CNN) em aplicações de reconhecimento de imagens [5].

Este trabalho, que é fruto do Programa Institucional de Bolsas de Iniciação Científica (PIBIC) realizado nos últimos 12 meses, propõe o desenvolvimento de uma arquitetura para um *hardware* que calcula a distância euclidiana utilizando técnicas de paralelismo cuja finalidade de acelerar o algoritmo *K-means*, uma vez que, dependendo do tamanho do conjunto de dados e da quantidade de características de cada elemento, o algoritmo pode demorar muito tempo para ser executado em um programa tradicional. Suas principais contribuições são: (1) o desenho de uma arquitetura paralela de *hardware* para servir de acelerador da distância euclidiana para o processamento de grande quantidade de dados; (2) uma descrição detalhada dos módulos implementados no *hardware* para a replicação do trabalho e (3) uma análise da aceleração com base nos resultados de testes simulados e cálculos matemáticos para *hardware* reconfigurável em tempo de execução.

2. CONTEXTUALIZAÇÃO

2.1. Referencial Teórico

O algoritmo *K-means* é uma técnica de aprendizado de máquina não supervisionada amplamente utilizada na análise de agrupamento de dados. Ele é usado para dividir um conjunto de dados em grupos ou *clusters* com base em suas similaridades. É frequentemente utilizado em várias aplicações, tais como:

- Segmentação de Clientes: Empresas podem usar o *K-means* para agrupar clientes com base em seu comportamento de compra, permitindo campanhas de *marketing* direcionadas.
- Análise de Imagem: Na área de processamento de imagem, o *K-means* pode ser usado para segmentar uma imagem em diferentes regiões com características de cores semelhantes.
- Bioinformática: O *K-means* é aplicado na *cluster*ização de genes com expressões gênicas semelhantes para identificar padrões em dados genéticos.
- Detecção de Anomalias: Ele também é usado na detecção de anomalias, onde *clusters* normais são estabelecidos e dados que não se encaixam bem nesses *clusters* são identificados como anomalias.

Como principais características do algoritmo *K-means*, listamos:

- Centroides: O algoritmo *K-means* agrupa os dados calculando centroides. Um centroide é uma representação do centro de um *cluster*. Geralmente, os iniciais são escolhidos aleatoriamente.
- Medida de Similaridade: A medida de similaridade mais comum usada pelo *K-means* é a distância euclidiana (existem outras como Mahalanobis e Manhattan por exemplo). A distância euclidiana calcula a distância entre dois pontos no espaço n-dimensional, onde n é o número de características ou dimensões dos dados. A fórmula da distância euclidiana entre dois pontos Q e K é (1):

$$d(Q, K) = ((Q_1 - K_1)^2 + (Q_2 - K_2)^2 + \ \dots \ + (Q_n - K_n)^2)^{1/2} \quad (1)$$

  Onde Q_1, Q_2, ..., Q_n e K_1, K_2, ..., K_n são as coordenadas do indivíduo (Q) e do centroide (K).
- Atribuição de Dados a *Clusters*: O algoritmo *K-means* atribui cada ponto de dados ao *cluster* cujo centroide está mais próximo, com base na medida de similaridade escolhida.
- Atualização dos Centroides: Depois de atribuir todos os pontos de dados aos *clusters*, o algoritmo calcula os novos centroides para cada *cluster*. Os novos centroides são calculados tomando-se a média das coordenadas de todos os pontos no *cluster*.
- Iteração: Os passos de atribuição e atualização de centroides são repetidos iterativamente até que os centroides não mudem significativamente ou um número predefinido de iterações seja alcançado.

Para a implementação do *K-means* é necessário observar o Algoritmo 1, em pseudocódigo, abaixo:

Algoritmo 1: Algoritmo *K-means*

Entrada: base de dados com i instâncias e d dimensões, K grupos desejados
Saída: Base de dados dividida em K grupos

```
1. início
2. inicializa os K centroides com valores aleatórios;
3. enquanto critério de parada não atingido;
4.    para cada amostra x_i faça
5.       Adicione x_i ao grupo do centroide c_k de menor distância, de acordo
   com a equação (1);
6.    fim
7.    Atualiza os centroides c_k de acordo com a equação (média das distâncias
   de cada grupo);
8. fim
```

Implementar um acelerador para o cálculo da distância euclidiana no *K-means* é uma escolha estratégica em função de várias razões:

- Demanda Computacional: O cálculo da distância euclidiana é uma parte fundamental do algoritmo *K-means* e é realizada repetidamente durante a execução do algoritmo, principalmente na etapa de atribuição de pontos aos *clusters*. Isso significa que, em termos de demanda computacional, é uma das partes mais intensivas em termos de processamento do algoritmo.
- Oportunidades de Paralelismo: O cálculo da distância euclidiana oferece excelentes oportunidades para a exploração do paralelismo. Cada cálculo de distância entre um indivíduo e um conjunto de centroides pode ser feito independentemente dos outros indivíduos, permitindo que as operações sejam paralelizadas em *hardware*, resultando em ganhos significativos de desempenho.
- Eficiência Geral do Algoritmo: Melhorar o desempenho do cálculo da distância euclidiana impacta diretamente a eficiência geral do algoritmo já que essa parte é realizada frequentemente. Qualquer otimização aqui se refletirá em melhoria significativa no tempo de execução do *K-means* como um todo.

- Benefício para Outras Aplicações: Além de acelerar o *K-means*, a implementação de um acelerador para o cálculo da distância euclidiana pode ser útil em outras aplicações de aprendizado de máquina e ciência de dados que dependem de cálculos de distância, como algoritmos de vizinhos mais próximos (K-NN), algoritmos de *clustering* hierárquico e muito mais.

2.2. Trabalhos Relacionados

O interesse em soluções de aceleração de algoritmos focadas em processamento de dados para tarefas de aprendizagem de máquina estimulou diversas pesquisas e trabalhos nesta área.

Sun e Cheng [6] propuseram uma arquitetura para sistemas embarcados baseado em *Artificial Neural Networks* (ANN) utilizando FPGA para classificação de doenças cardíacas, tendo como base as informações fornecidas pelo eletrocardiograma (ECG). A implementação em FPGA foi 5000 vezes mais rápida e gastou 4000 vezes menos energia que sua implementação feita utilizando *smartphones*. Os autores concluem também que é possível uma redução de 98,7% de utilização de área, 99,1% de economia de energia dependendo da arquitetura utilizada, porém com o sacrifício do tempo de resposta.

O trabalho de [7] apresenta uma solução heterogênea de *hardware* e *software* para aceleração do cálculo de distância euclidiana ao quadrado para o algoritmo *K-means*. O acelerador em *hardware* foi validado por meio de simulações em situações distintas utilizados dados sintéticos, podendo constatar o atendimento de requisito funcional, em relação a corretude, mas também sugerindo um ganho de desempenho em relação a uma versão sequencial. Pelas características de semelhança esse trabalho serviu de semente para o nosso trabalho.

A aplicação de [8] apresenta uma solução híbrida que calcula o algoritmo *K-means* e sua variação para dados categóricos, K-modes. Esta solução permite configurar os parâmetros em tempo de execução, sem necessidade de recompilar o código de descrição do dispositivo.

Em [9], a arquitetura recebe entradas de 64 bits e suporta dados em ponto fixo ou flutuante. Os parâmetros podem ser ajustados em tempo de operação do *hardware,* porém a memória é carregada em tempo de compilação, sendo necessário recompilar o projeto em caso de substituição da base de dados.

As duas arquiteturas apresentadas anteriormente não exploram o paralelismo quanto poderiam, processando apenas duas informações ou dimensões por ciclo.

O trabalho [10] utiliza PYNQ (AMD/Xilinx) e o ambientes Jupyter Notebook para criar uma solução híbrida *hardware /software*. Para a parte *hardware,* é utilizada a síntese de alto nível (HLS) que permite que o *hardware* seja descrito em C++ por meio de diretivas de compilação. Ao optar por usar HLS, o desenvolvimento é simplificado, mas o controle sobre a construção de *hardware* que o design da camada de transferência de registro (RTL) permite é perdido. Os testes mostram que a versão construída em RTL é duas vezes mais rápida que a versão construída com HLS, porém essa versão ainda é vinte e quatro vezes mais rápida comparada a um sistema com duas CPUs Xeon Silver 4210R. Esse projeto não permite a modificação dos parâmetros em tempo de execução.

Outro trabalho que utiliza uma linguagem de alto nível é [11] que apresenta uma solução utilizando a biblioteca *Veriloggen*, *framework* que permite fazer a descrição do *hardware* em Python. Nessa arquitetura o cálculo da distância entre os indivíduos e os centroides foi completamente paralelizado, mas todos os ajustes de parâmetros são feitos em tempo de compilação.

O trabalho de [12] traz uma abordagem mais focada na paralelização dos processos do *K-means*. Essa arquitetura traz o parâmetro G que define o grau de paralelização que o *hardware* terá, em tempo de compilação. Esse parâmetro define quantos pontos de dados serão processados por vez. Durante o cálculo de distância, todas as dimensões dos G pontos de dados são processados junto com todas as dimensões de todos os centroides a cada ciclo. Os demais parâmetros também são definidos na compilação. Os testes desse trabalho mostram que a paralelização completa compensa a grande ocupação dos recursos do FPGA com o ganho na velocidade do processamento dos dados.

3. METODOLOGIA E DESENVOLVIMENTO

Desenvolver um *hardware* acelerador para funcionar como um coprocessador de uma aplicação específica é um processo complexo que envolve várias etapas. A primeira etapa é o desenho da arquitetura: para isso é preciso identificar claramente os requisitos da aplicação que o acelerador deve atender, o que inclui entender as operações que precisam ser aceleradas, as restrições de desempenho, os padrões de acesso à memória e outros aspectos importantes para só então projetar a arquitetura do *hardware* propriamente, com seus blocos funcionais necessários, a interconexão entre eles e a identificação das oportunidades de paralelismo. O desenho deve levar em consideração as características específicas da aplicação e os desafios de sincronização.

Depois da etapa primordial de desenho da arquitetura do *hardware*, outras etapas importantes incluem:

- Seleção de Tecnologia: Escolher a tecnologia apropriada, como FPGAs, ASICs ou outras personalizadas, com base nas necessidades da aplicação e recursos disponíveis.
- Implementação em HDL: Traduzir o desenho da arquitetura para uma linguagem de descrição de *hardware* (HDL), como VHDL ou Verilog.

- Simulação e Testes: Realizar simulações para verificar o funcionamento correto do desenho em HDL, corrigindo problemas de lógica ou desempenho quando necessário.
- Síntese: Transformar o desenho em HDL numa implementação física, como FPGA.
- Testes em Placa de Desenvolvimento: Testar o *hardware* em um ambiente próximo ao de produção.
- Otimização de Desempenho: Realizar testes de desempenho e otimizar o *hardware*, fazendo ajustes na arquitetura, alocação de recursos e código em HDL.
- Integração com a Aplicação: Integrar o *hardware* à aplicação, adaptando-a para usar eficazmente o coprocessador.
- Teste e Validação: Realizar testes abrangentes para validar o funcionamento e o desempenho do sistema integrado (aplicação + *hardware*).
- Documentação e Manutenção: Documentar o processo de desenvolvimento, incluindo design, implementação, simulações e resultados de desempenho, para futuras melhorias e manutenção.
- Implantação em Produção: Após validação bem-sucedida, implantar o *hardware* em ambiente de produção, onde ele será usado como coprocessador pela aplicação.

Para realizar o desafio proposto, o desenho da arquitetura do *hardware* acelerador, fizemos uso da ferramenta Logisim (versão 2.7.1) [13]. O Logisim é uma ferramenta que possui uma interface de usuário intuitiva e baseada em componentes gráficos, o que torna o design de circuitos digitais mais acessível, possui ainda um editor de desenho de circuitos, que permite desenhar circuitos digitais usando componentes lógicos padrão, como portas lógicas, *flip-flops*, registradores, multiplexadores, entre outros. Além disso, oferece a capacidade de simular circuitos em tempo real. Os usuários podem aplicar entradas aos circuitos e observar as saídas em resposta às mudanças nas entradas. Com o uso da ferramenta foi possível não apenas projetar, mas também simular os circuitos digitais que compõem nossa solução.

O módulo identificado como DM na Figura 1 é o responsável por realizar todo o cálculo da distância euclidiana [14], descrito aqui pela seguinte equação (2):

$$d = \sqrt{\sum_{i=1}^{i=N} \left(x_{Qi} - x_{Ki}\right)^2} \quad (2)$$

sendo x_{Qi} e x_{Ki} as características do indivíduo e do centroide, e N o número total de características.

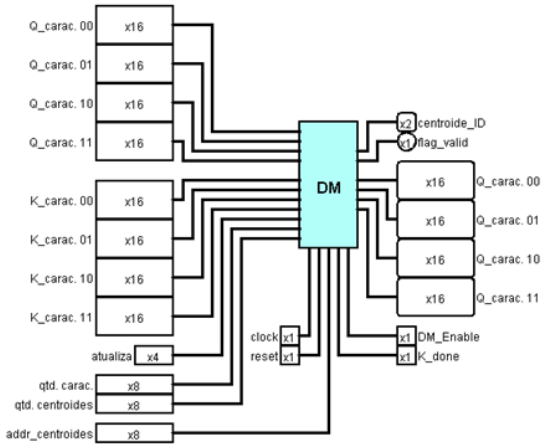

Figura 1 - Módulo Medidor de Distância (Autoria própria)

A partir dos dados de entrada, usados para armazenar os indivíduos com até quatro características e até quatro centroides iniciais, representados na Figura 1 pelos pinos com valores binários, cada indivíduo tem sua distância para cada um dos centroides calculada de forma simultânea e, como resultado retorna, além da distância, o identificador do centroide que correspondeu a menor distância calculada (centroide_ID), o indivíduo que foi analisado, e uma *flag* (DM_done) avisando o fim do cálculo. Essas informações são necessárias para a atualização dos centroides, conforme o algoritmo do *K-means*. As *flag*s K_done e DM_Enable informam que o conjunto de características dos centroides fora carregado e o circuito está pronto para calcular respectivamente. A porta "atualiza" será usada posteriormente para informar quais centroides serão atualizados.

A Figura 2 apresenta os detalhes internos da arquitetura proposta, a qual possui diversos submódulos, que realizam diferentes etapas do processo. Nessa figura foi utilizada um recurso da ferramenta Logisim chamada de túnel, a qual permite ligar dois ou mais componentes utilizando túneis de mesmo nome e evitar muitos fios, para facilitar a visualização. Os túneis são representados por uma seta com um nome. Assim, para conectar dois ou mais circuitos basta utilizar túneis com mesmo nome. Neste projeto, os túneis foram usados para ligar as entradas das características dos indivíduos (Q_caract. 00, ..., Q_caract. 11), as dos centroides (K_caract. 00, ..., K_caract. 11), o *clock* (clk), reset, entre outros. Vamos usar como exemplo a entrada *"reset"* a qual está ligada ao túnel "rst" e que também aparece em outras partes como nas entradas dos blocos "temp", "OP", registradores, entre outros.

Nesta arquitetura os dados de entrada referente as características dos centroides são primeiramente armazenadas nos submódulos “temp”. As características dos indivíduos que precisam ser calculados chegam por streaming e, portanto, são enviadas diretamente para os blocos que fazem os cálculos de distância propriamente dito (OP). Cada submódulo OP faz o cálculo de distância de um centroide, logo, as saídas de cada um deles são conectadas as entradas de submódulos comparadores (COMP) em forma de árvore até a comparação da menor distância, disponibilizada na saída da arquitetura.

Deve-se notar que os submódulos de *hardware* que não possuem dependência de dados trabalham em paralelo entre si, sendo responsável pelo primeiro nível de paralelismo dessa arquitetura. O segundo nível é o de *pipeline*, ou seja, os dados fluem da esquerda para direita, conforme a Figura 2, e cada submódulo faz seu trabalho salvando seus resultados em registrador para ser acessado pelo submódulo seguinte em um ciclo de *clock* posterior. Desse modo, no ciclo seguinte o submódulo está apto a processar novos dados enquanto o submódulo sucessor trabalha em paralelo, mas no primeiro dado. Portanto, a aceleração da arquitetura deve-se a estes dois níveis de paralelismo. A seguir, serão apresentados os detalhes dos submódulos.

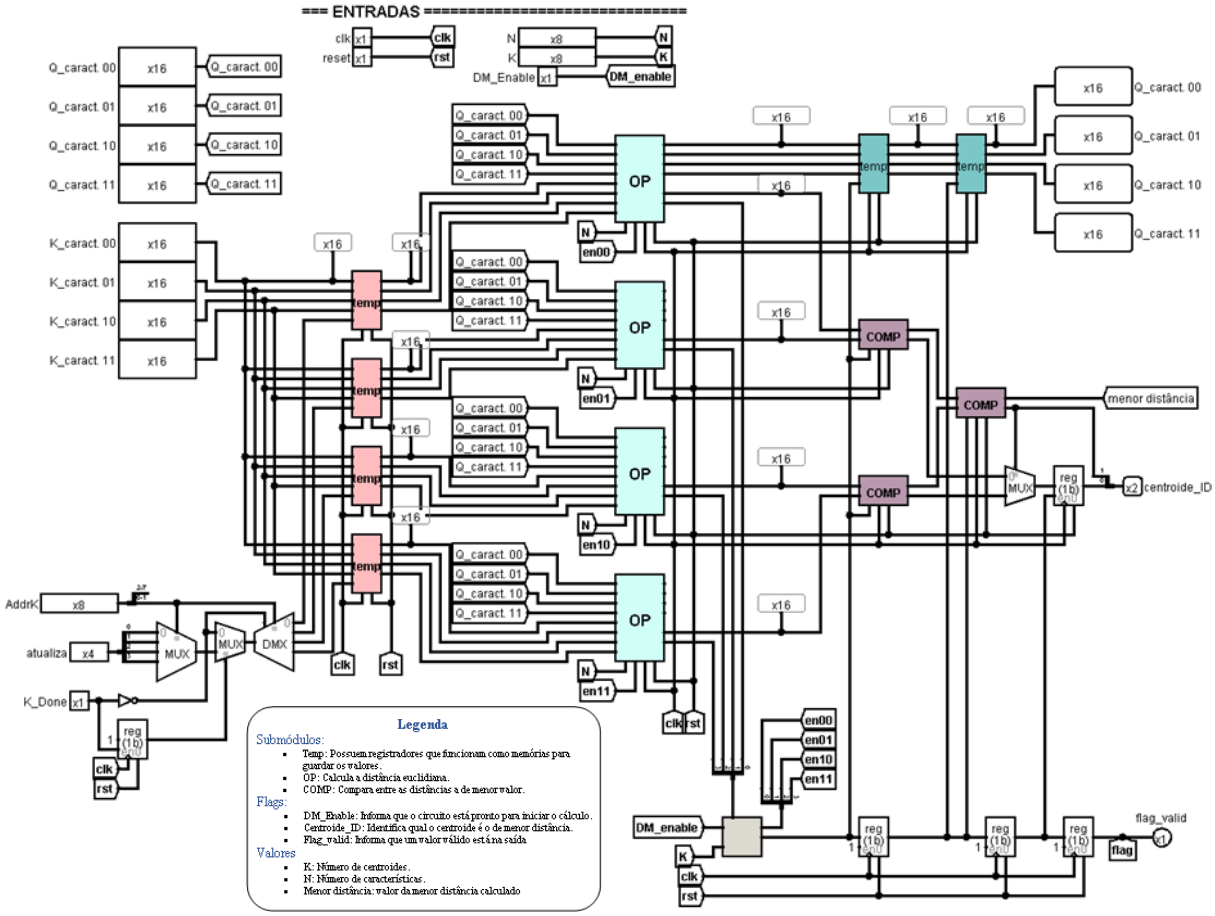

Figura 2 – Circuito Medidor de Distância (Autoria própria)

3.1 Operadores:

Os operadores, identificados como OP na Figura 2, são blocos utilizados para realizar o cálculo de distância propriamente dito. Os detalhes do submódulo OP são apresentados na Figura 3.

Dentro deles, as distâncias de cada característica do indivíduo e do centroide são calculadas individualmente, também em paralelo de acordo com a quantidade de centroides configurada, sendo a primeira etapa o cálculo da diferença ao quadrado (QUAD), seguido pela “árvore de somas” e pela “raiz quadrada” (RAIZ) os quais serão apresentados em seguida. A árvore de soma é um conjunto de dois somadores que trabalham em paralelo e recebem os resultados dos submódulos QUAD, em seguida um terceiro somador adiciona os resultados dos somadores anteriores. A profundidade dessa árvore é proporcional à quantidade de QUADs, que por sua vez depende do número de características computadas em paralelo. A saída do submódulo OP é o valor da menor distância, as características do indivíduo e a *flag* OP_done, que indica que o cálculo foi concluído.

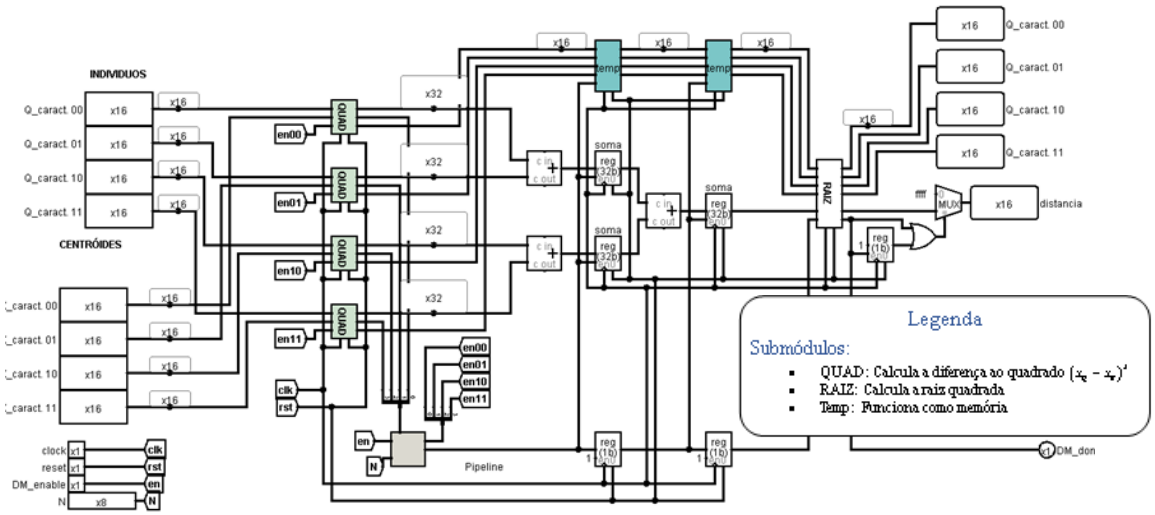

Figura 3 - Circuito Operadores (Autoria própria)

3.2 Diferença ao quadrado:

Dando início ao cálculo da distância propriamente, os valores das características dos indivíduos e dos centroides são recebidos pelos módulos de ***"diferença ao quadrado"*** (QUAD), que irá calcular o quadrado da diferença de cada característica em relação ao centroide $(x_Q - x_K)^2$. A quantidade de módulos QUAD ativos é igual ao número de características (N) de centroides e indivíduos, todavia, como eles trabalham em paralelo e temos no máximo 4 características, são necessários apenas 3 ciclos de *clocks* para chegar ao primeiro resultado das distâncias de todas as características simultaneamente. Depois disso, o *pipeline* de três estágios garante um novo resultado a cada ciclo de *clock.* Os detalhes de implementação do QUAD são apresentados *na* Figura 4.

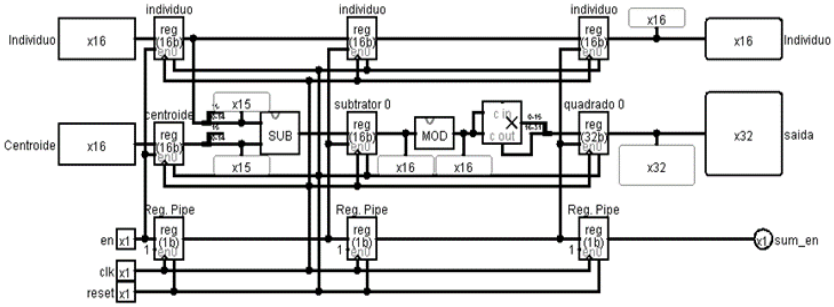

Figura 4 - Circuito QUAD (Autoria própria)

3.3 Raiz quadrada:

O cálculo é feito usando recursos de Cálculo Numérico, baseado no Método de Heron (Alexandria-Egito 10 d.C. – 70 d.C.) [15] demonstrado no Algoritmo 2, descrito abaixo:

Algoritmo 2: Raiz quadrada
Entrada: S, n
Saída: $\sqrt{S}$

```
1. início
2. Y = 1 + (½S);
3. se (int i = 0; i < n; i++) faça
4.     X_i = ½(Y + S/Y)
5.     Y = X_i;
6.     fim
7.     imprima (Y)
8. fim
```

Sendo S o número que se quer extrair a raiz quadrada, n a quantidade de iterações. Uma implementação em *software* está disponível no link[1].

[1] Código do método de Heron:
https://colab.research.google.com/drive/1H7rCsQ8ChmwIeTooUKH7VyIRYTce0VwW#scrollTo=9NvJ1MQIqjAe

A implementação do *hardware* que calcula a raiz quadrada corresponde a um *pipeline* de dez estágios para obter uma boa precisão numérica do resultado com números de 16 bits. A Figura 5, em corte mostra apenas quatro destes estágios e cada um deles, exceto o primeiro, é composto por uma divisão, uma soma e um shift para a direita (o mesmo que dividir por dois).

Uma consideração importante que depreende é a necessidade de elevar ao quadrado a diferença entre o indivíduo e o centroide, ou seja, é necessário realizar uma multiplicação de um valor por ele mesmo. Um *hardware* que multiplica um número de m bits por outro de n bits tem um resultado em até $m + n$ bits. Sendo assim, o *hardware* que calcula o quadrado deve ter entrada de um número de p bits e saída de $2 \times p$ bits, caso contrário há perda de informação ou inconsistência na largura de bits entre o elemento (p bits) e do centroide ($2 \times p$ bits) na euclidiana simples (sem a raiz quadrada). A utilização da distância euclidiana resolve esse problema, uma vez que o somatório das diferenças ao quadrado de $2 \times p$ bits serve de entrada para uma raiz quadrada, que gera uma saída de p bits, assim como os dados de entrada do cálculo de distância.

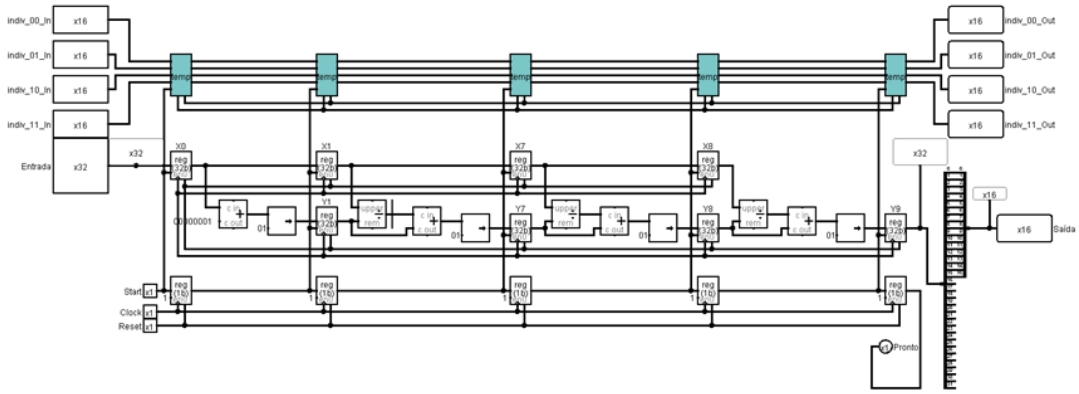

Figura 5 - Circuito Raiz Quadrada Pipeline (Autoria própria)

3.4 Árvores de comparadores:

Por fim, após o indivíduo ter passado por esse processo para cada centroide, todas as distâncias calculadas são comparadas entre si, por meio da "árvore de comparadores" (COMP), a qual corresponde a dois comparadores em paralelo que alimentam um terceiro comparador, tal qual a "árvore de somas". De forma análoga, a profundidade dessa árvore é determinada pela quantidade máxima de centroides paralelos. Essa árvore compara os resultados de distância dois a dois em paralelo, usando um pipeline de dois estágios, até encontrar a menor distância calculada, indicando ainda o centroide "vencedor", em que se verificou essa distância. Os detalhes do circuito são apresentados na Figura 6.

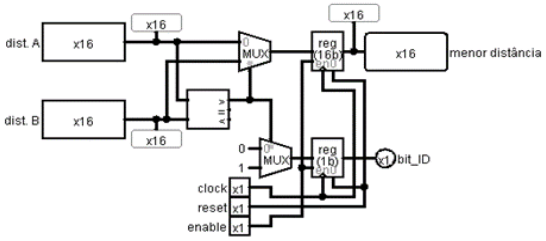

Figura 6 - Circuito Comparador (Autoria própria)

4. RESULTADOS E DISCUSSÕES

Este trabalho corresponde a etapa de design da arquitetura e, portanto, ainda não foi implementada em uma linguagem de descrição de *hardware,* nem foi mapeada para uma tecnologia específica como uma família de FPGAs ou de transístores. No entanto, vale salientar a importância dessa etapa como a validação da arquitetura e de modo a direcionar as próximas etapas de implementação para caminhos mais promissores. Durante a investigação das escolhas de projeto foi possível fazer diversos refinamentos até chegar ao modelo atual. Portanto, como forma de avaliação da arquitetura projetada foram obtidos resultados de simulação por meio do Logisim (versão 2.7.1). Para isso implementamos um circuito de teste (*testbench*) em que os dados (características) dos centroides iniciais e dos indivíduos são enviados por streaming, ou seja, de forma sequencial ininterruptamente a cada ciclo de *clock* para a arquitetura proposta para cálculo da distância euclidiana. Nesses *testbenches* utilizamos circuitos de memórias RAM, as quais podem ser inicializadas com arquivos no formato específico do Logisim. Dessa forma, geramos os arquivos de inicialização das memórias por meio de uma implementação do *K-means* em Python também para servir de base de comparação com as simulações do *hardware* no Logisim.

Assim, as simulações foram usadas para avaliar: (1) a corretude do algoritmo implementado em *hardware*; (2) análise das características de aceleração por meio do paralelismo da arquitetura proposta; (3) a aceleração potencial da arquitetura em comparação a implementação do algoritmo puramente em *software*.

4.1 Corretude

Para avaliação da confiabilidade da arquitetura como um circuito que calcula corretamente a distância euclidiana conforme sua definição apresentada na seção 2.1, foram utilizados dois conjuntos de dados (*datasets*) diferentes. O primeiro é um *dataset* sintético, a qual distribui de forma uniforme 128 indivíduos com 2 características nos quatro quadrantes de um plano cartesiano. Após a *clusterização* em 4 centroides os indivíduos devem ficar distribuídos conforme a Figura 7(a). A implementação em *software* está disponível no link[2].

No *testbench* do circuito da arquitetura foram incluídos um circuito comparador e um contador. O comparador recebe o resultado produzido pela arquitetura e o valor esperado armazenado em uma memória inicializada com dados originados pela aplicação em Python e liga uma *flag* quando os resultados são iguais. Tal *flag* aciona o contador para contar quantos resultados foram produzidos corretamente. Nesse teste com o primeiro *dataset*, o sintético, como os dados eram bem separados não houve diferença do resultado esperado.

No segundo experimento de corretude foi utilizado o conhecido *dataset* Iris da biblioteca Scikit-learn, o qual contém 150 indivíduos com 4 características cada. Diferentemente do teste anterior os indivíduos agora estão mais sobrepostos, como pode ser visto na Figura 7(b) que plota 1 par de características dos indivíduos em 4 *clusters*. Vale salientar que os dados originais desse *dataset* estão em ponto flutuante com 1 casa decimal, no entanto, como o *hardware* proposto trabalha apenas com representação inteira, todos as características foram multiplicadas por 10 para gerar os arquivos que inicializam as memórias do *testbench* da arquitetura.

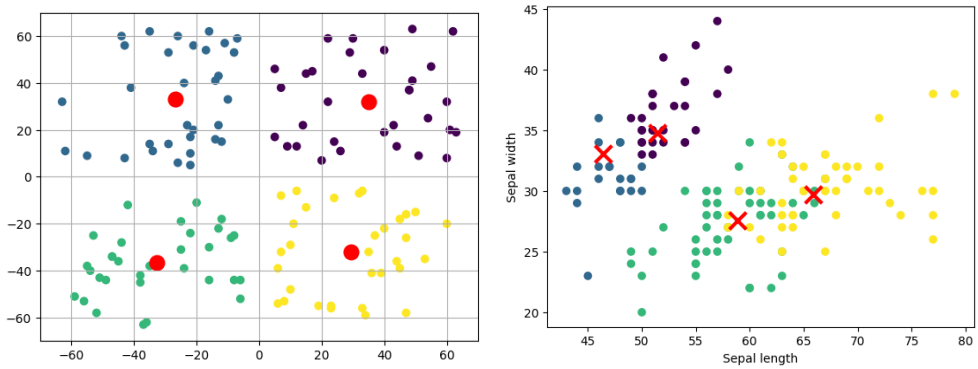

Figura 7 – Quatro *clusters* e seus centroides: (a) *dataset* sintético; (b) *dataset* Iris (Autoria própria)

Em comparação com a implementação em *software*[3] houve vinte e seis divergências, sendo que as distâncias calculadas pelo circuito estavam corretas. Desses 26 indivíduos, 10 tiveram a menor distância calculada igual para mais de um centroide e, nesse caso, o circuito por padrão o atribui ao primeiro centroide de menor valor enquanto que o algoritmo da biblioteca Scikit-learn, que calcula em ponto decimal, tomou de decisão diferente. Outras oito distâncias divergentes estão muito próximas, podendo a diferença estar na questão de arredondamento por parte do *software* enquanto o circuito despreza os números depois da vírgula. As demais se devem ao fato do *K-means* em *software* ter apresentado resultados inconsistentes. As distâncias dos vinte e seis indivíduos para os quatro centroides no circuito são sumarizadas na Tabela 1 nas 4 primeiras colunas. Na coluna "Class. SW" é como a implementação em *software* atribuiu e na "Class. HW" como o *hardware* proposto o fez. Para uma compreensão mais detalhada do que chamamos "resultados inconsistentes do *K-means* em *software*" veja um exemplo no Apêndice A.

Tabela 1 - Distâncias dos indivíduos divergentes do *dataset* Iris

Indivíduo	**D0**	**D1**	**D2**	**D3**	***Cluster* SW**	***Cluster* HW**
1	5	6	34	45	1	0
3	6	6	35	45	1	0
9	4	5	34	44	1	0
11	3	3	33	44	1	0
12	6	6	35	45	1	0
13	9	9	40	50	1	0
25	5	6	32	43	1	0
29	5	5	33	44	1	0
30	5	5	33	43	1	0
34	4	5	33	44	1	0

[2] Código do *dataset* sintético:
https://colab.research.google.com/drive/1VJdMhiOQR5ZslOzs6kUEVs6q8Di61qLr#scrollTo=x8aE4lgGnqqB
[3] Código do *dataset* Íris:
https://colab.research.google.com/drive/1ZbmQvtjsDBjnk_jEum_O3N0UsA_wFNPn#scrollTo=VvpYi3ZY0LIV

35	3	4	35	46	1	0
37	2	0	35	46	0	1
41	13	13	37	47	1	0
45	6	6	34	45	1	0
47	5	5	36	46	1	0
50	40	41	7	11	3	2
52	41	42	8	8	3	2
70	38	39	9	8	2	3
72	40	41	7	6	2	3
76	40	41	6	8	3	2
86	39	40	5	8	3	2
113	41	42	12	7	2	3
119	41	42	10	8	2	3
121	40	41	12	9	2	3
126	39	40	7	5	2	3
138	38	40	8	7	2	3

4.2 Análise do paralelismo da arquitetura

Embora a arquitetura possua diversos recursos de paralelismo, o *hardware* deve funcionar como um acelerador de uma aplicação em *software*, a qual deve fornecer os valores dos dados iniciais de indivíduos e centroides. Logo, esses dados entram no bloco de cálculo da distância de forma sequencial. Assim, a arquitetura é responsável por armazenar parte dos dados temporariamente para só então explorar o paralelismo real e o pipeline. Portanto, para o cálculo do primeiro indivíduo há uma demora maior. Essa demora corresponde a carga dos centroides, do primeiro indivíduo (em número igual ao de características) e o percurso total do circuito Medidor de Distância (DM). A partir do segundo indivíduo começa-se a haver aceleração.

Para o cálculo do primeiro indivíduo que chega ao acelerador é necessário que ele atravesse todo o circuito, garantindo o paralelismo apenas da comparação de todas as suas características com todos os centroides. Portanto, a Tabela 2 apresenta a quantidade de ciclos de *clock* que cada circuito necessita para formar o caminho de dados para cálculo de um indivíduo e assim formar o *pipeline*. Logo, a profundidade do pipeline é dada pelo somatório desses ciclos, o qual corresponde a 17 estágios, sendo este o potencial de aceleração do pipeline, além do paralelismo de primeiro nível.

Tabela 2 - Quantidade de siclos por módulo

Circuito	Ciclos	Obs.
Distância quadrada	3	Subtração + MOD + Quadrado
Árvore de Soma	2	Log_2(Qtd. Máxima de características)
Raiz Quadrada	10	Raiz calculada por método numérico
Árvore de comparadores	2	Log_2(Qtd. Máxima de centroides)
Total	17	

Embora a quantidade de características e de centroides nesta arquitetura sejam configuráveis, a quantidade de ciclos da árvore de soma e de comparadores possuem valores fixos pois são definidos pela quantidade máxima desses parâmetros. Assim, caso os valores dos parâmetros sejam menores que os máximos, alguns circuitos paralelos não geram resultados válidos, portanto sendo desprezados, mas mantendo a quantidade de ciclos nesses circuitos em árvore.

Para realizar os testes a seguir foram usados dados aleatórios com 4 características (valor de N), 4 centroides (valor de K) e um número de indivíduos (Q) variando de 4 a 512, conforme a Tabela 3.

Tabela 3 - Valores simulados no Logisim

Teste	N	K	Q	Carga K	Carga Q	Carga Total	Primeiro resultado	Último resultado	Sequencial	Eficiência	%	*Clock*/I
1	4	4	4	16	16	32	37	49	148	3,020408	66,9	12,2500
2	4	4	8	16	32	48	37	65	296	4,553846	78,0	8,1250
3	4	4	16	16	64	80	37	97	592	6,103093	83,6	6,0625
4	4	4	32	16	128	144	37	161	1184	7,354037	86,4	5,0313
5	4	4	64	16	256	272	37	289	2368	8,193772	87,8	4,5156
6	4	4	128	16	512	528	37	545	4736	8,689908	88,5	4,2578
7	4	4	256	16	1024	1040	37	1057	9472	8,961211	88,8	4,1289
8	4	4	512	16	2048	2064	37	2081	18944	9,103316	89,0	4,0645
9	4	4	1024	16	4096	4112	37	4129	37888	9,176072	89,1	4,0322
10	4	4	2048	16	8192	8208	37	8225	75776	9,212888	89,1	4,0161
11	4	4	4096	16	16384	16400	37	16417	151552	9,231406	89,2	4,0081

12	4	4	8192	16	32768	32784	37	32801	303104	9,240694	89,2	4,0040
13	4	4	16384	16	65536	65552	37	65569	606208	9,245345	89,2	4,0020
14	4	4	32768	16	131072	131088	37	131105	1212416	9,247672	89,2	4,0010
15	4	4	65536	16	262144	262160	37	262177	2424832	9,248836	89,2	4,0005

A coluna “Carga K” da Tabela 2, indica a quantidade de ciclos para receber todos os centroides, ou seja, para 4 centroides com 4 características (Carga K = Qtd. Centroides × Qtd. Características), é necessário 16 *clock*s em todos os cenários. O mesmo vale para a quantidade de *clock*s para carregar os indivíduos (128 *clock*s), ou seja, o produto da quantidade de indivíduos pela quantidade de características (Carga Q = Qtd. Indivíduos × Qtd. Características). A quantidade de *clock*s para carregar todos os dados, centroides e indivíduos, é a soma das cargas de centroides e indivíduos (Carga total = Carga K + Carga Q).

O primeiro resultado é apresentado depois de carregado todos os centroides e o primeiro indivíduo carregado atravessar todo o circuito, ou seja, percorrer os 17 *clock*s da Tabela 1, o que nesse exemplo acontece no *clock* de número 37 (Primeiro resultado = Carga Centroides + Qtd. Características + Qtd. *clock*s Medidor de Distância). A partir deste ponto vamos tendo um novo resultado a cada 4 *clock*s, sendo o último resultado apresentado no *clock* 161 (Último resultado = Primeiro resultado + (Qtd. Indivíduos - 1) × Qtd. Características). Os valores apresentados na tabela correspondem aos valores teóricos definidos por estas equações, os quais foram confirmados nas simulações no Logisim, com contagem de ciclos.

Para verificar a eficiência do pipeline, foi incluída na Tabela 3 a coluna “Sequencial” que apresenta a quantidade de ciclos do circuito sem pipeline (coluna sequencial) para as mesmas configurações. As colunas seguintes “Eficiência” e “%” apresentam respectivamente a melhoria absoluta e percentual da versão pipeline em comparação ao circuito sequencial. Percebe-se que a eficiência vai aumentando com o volume de dados até se estabilizar em 9 vezes mais eficiente a partir do teste 8. E por fim, a última coluna apresenta a média de ciclos por indivíduo, a qual vai diminuindo e se estabilizando a partir do teste 5.

Tabela 4 - Valores calculados por Qtd. Indivíduos

Teste	**N**	**K**	**Q**	**Carga K**	**Carga Q**	**Carga Total**	**Primeiro resultado**	**Último resultado**	**Sequencial**	**Eficiência**	**%**	***Clock*/I**
1	4	4	4	16	16	32	37	49	148	3,020408	66,9	12,2500
2	4	4	8	16	32	48	37	65	296	4,553846	78,0	8,1250
3	4	4	16	16	64	80	37	97	592	6,103093	83,6	6,0625
4	4	4	32	16	128	144	37	161	1184	7,354037	86,4	5,0313
5	4	4	64	16	256	272	37	289	2368	8,193772	87,8	4,5156
6	4	4	128	16	512	528	37	545	4736	8,689908	88,5	4,2578
7	4	4	256	16	1024	1040	37	1057	9472	8,961211	88,8	4,1289
8	4	4	512	16	2048	2064	37	2081	18944	9,103316	89,0	4,0645
9	4	4	1024	16	4096	4112	37	4129	37888	9,176072	89,1	4,0322
10	4	4	2048	16	8192	8208	37	8225	75776	9,212888	89,1	4,0161
11	4	4	4096	16	16384	16400	37	16417	151552	9,231406	89,2	4,0081
12	4	4	8192	16	32768	32784	37	32801	303104	9,240694	89,2	4,0040
13	4	4	16384	16	65536	65552	37	65569	606208	9,245345	89,2	4,0020
14	4	4	32768	16	131072	131088	37	131105	1212416	9,247672	89,2	4,0010
15	4	4	65536	16	262144	262160	37	262177	2424832	9,248836	89,2	4,0005

É possível ainda calcular e inferir que fixando o número de indivíduos e variando o número de centroides (Tabela 5) há um aumento do número de *clock*s pouco significativo, o que faz sentido visto que a arquitetura compara cada indivíduo em paralelo com todos os centroides.

Tabela 5 - Variação de Qtd. Centroides

Teste	**N**	**K**	**Q**	**Carga K**	**Carga Q**	**Carga Total**	**Primeiro resultado**	**Último resultado**	**Qtd. *Clocks* /Ind**
16	4	2	1024	8	4096	4104	45	4137	4,04004
17	4	4	1024	16	4096	4112	53	4145	4,04785
18	4	8	1024	32	4096	4128	69	4161	4,06348
19	4	16	1024	64	4096	4160	101	4193	4,09473
20	4	32	1024	128	4096	4224	165	4257	4,15723

Também variamos o número de características (Tabela 6), mantendo os demais parâmetros. Neste caso houve uma relação direta de aumento mais significativo da quantidade de ciclos. Isso é mais facilmente percebido na Figura 12 com gráficos na mesma escala.

Tabela 6 - Variação de Qtd. Características

Teste	N	K	Q	Carga K	Carga Q	Carga Total	Primeiro resultado	Último resultado	Qtd. *Clocks* /Ind
21	2	4	1024	8	2048	2056	43	2089	2,04004
22	4	4	1024	16	4096	4112	53	4145	4,04785
23	8	4	1024	32	8192	8224	73	8257	8,06348
24	16	4	1024	64	16384	16448	113	16481	16,09473
25	32	4	1024	128	32768	32896	193	32929	32,15723

É importante observar que para o cálculo dos valores apresentados na Tabela 6 foi necessário mudar a quantidade de ciclos das árvores de comparadores e de somas de 2 para 5 estágios.

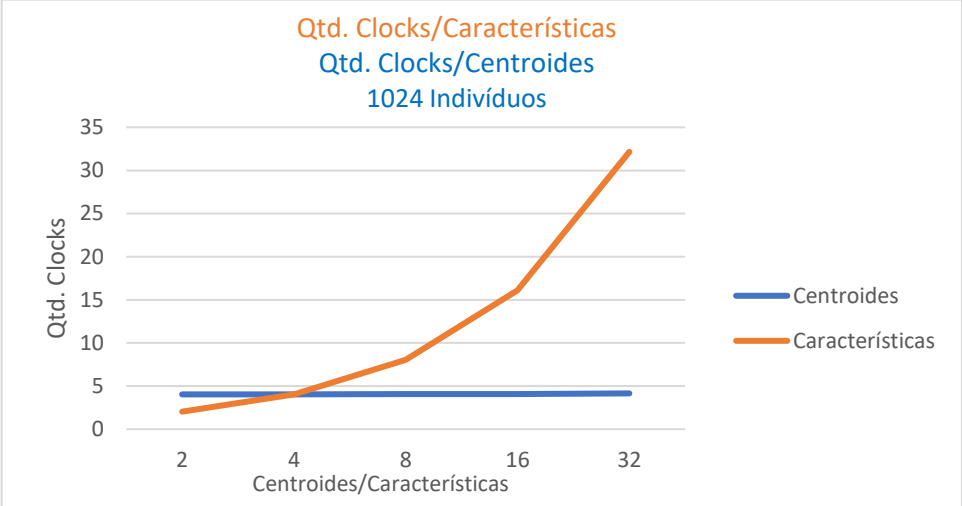

Figura 8 - Comparativos das variações de Centroides e Características (Autoria própria)

4.3 – Aceleração potencial

Para avaliar o potencial de aceleração da arquitetura projetada foi implementado o mesmo algoritmo de distância euclidiana em assembly[4] do processador MIPS, de modo que se pudesse contabilizar o número de instruções como métrica de comparação com o *hardware* proposto neste trabalho. A implementação em assembly é responsável por realizar o cálculo de distância euclidiana de 1 único indivíduo com quatro características para 4 centroides, os quais são comparados e indicado o *cluster* que tal indivíduo deve ser alocado.

A implementação assembly utilizou-se de pseudoinstruções, mas com a com ajuda do MARS [16], um assembler/simulador de MIPS, que traduz as pseudoinstruções para instruções básicas, obtivemos informações que nesta implementação há 27 instruções fora do laço de repetição e 9 dentro do laço. A quantidade de repetições do laço é determinada pela quantidade de características dos indivíduos, o que resultou em 61 instruções executadas. Como há 27 instruções fixas, podemos considerar que é necessário a execução de 34 instruções, correspondentes aquelas que estão dentro do laço. Para simplificar as comparações vamos considerar que todas as instruções executam em 1 único ciclo e que para um número maior de indivíduos, multiplicaremos a quantidade de indivíduos por 34 instruções.

Em todos os testes realizados nesta etapa utilizou-se de dados genéricos com 4 características e *cluster*ização em 4 grupos. Os mesmos dados da aplicação assembly foram utilizados na arquitetura proposta. Para comparação dos resultados a quantidade de indivíduos variou de 4 a 65.536, conforme a Tabela 7. Na coluna "# Instruções MIPS Executadas" são apresentadas a possível quantidade de instruções que o MIPS executa fazendo as considerações apresentadas anteriormente. Na coluna seguinte temos a quantidade de ciclos de *clock*, considerando que ele implemente um pipeline de 5 estágios, que é o padrão do processador MIPS. A terceira coluna apresenta a quantidade de ciclos na arquitetura proposta com as mesmas configurações. E por fim, a aceleração teórica da arquitetura em relação ao MIPS. Percebe-se que a aceleração aumenta com o crescimento do volume de dados, se estabilizando em 8 vezes mais rápido a partir de 256 indivíduos.

Tabela 7 - Comparativo Qtd. Instruções x Qtd. Clocks

# Indivíduos	# Instruções MIPS executadas	# Ciclos MIPS com pipeline	# *Clock*s Arq. proposta	Aceleração
4	163	167	53	3,08
8	299	303	69	4,33
16	571	575	101	5,65

[4] Código assembly: https://github.com/webminst/TCC/blob/main/Kmeans.asm

32	1115	1119	165	6,76
64	2203	2207	293	7,52
128	4379	4383	549	7,98
256	8731	8735	1061	8,23
512	17435	17439	2085	8,36
1024	34843	34847	4133	8,43
2048	69659	69663	8229	8,47
4096	139291	139295	16421	8,48
8192	499712	278559	32805	8,49
16384	999424	557087	65573	8,50
32768	1998848	1114143	131109	8,50
65536	3997696	2228255	262181	8,50

CONCLUSÕES

Este trabalho mostra um acelerador para distância euclidiana, cujo objetivo de aumentar o desempenho da execução do algoritmo *K-means*, projetado com auxílio da ferramenta Logisim. A partir de simulações e cálculos podemos constatar o atendimento de requisito funcional, em relação a correção, mas também podemos fazer inferências a respeito das variáveis que impactam no tempo de respostas do acelerador.

O grande ganho de desempenho do acelerador está no uso de paralelismo na comparação de indivíduos com centroide e uso de *pipeline* nas etapas do cálculo da distância, além do paralelismo empregado na árvore de soma e de comparação usada na implementação. No que diz respeito ao *pipeline*, o aumento da quantidade de indivíduos e iterações tornará a eficácia do acelerador ainda mais evidente em comparação a soluções puramente em sequenciais.

Comparados aos resultados de [7], em que o resultado do primeiro indivíduo necessitava 58 ciclos de *clock* e mais 10 ciclos para cada novo indivíduo, este se mostra ainda mais eficiente em condição semelhante (4 Centroides, 4 características), haja visto que precisa de apenas 37 ciclos de *clock* para o primeiro indivíduo e 4 para os demais mesmo tendo que calcular a raiz quadrada.

Como trabalhos futuros, o acelerador deverá ser aperfeiçoado para receber dados de até 32 bits, ter incluído um módulo para calcular a atualização dos centroides e a estrutura de controle com uso de uma máquina de estados e, por fim, ser convertido em linguagem de descrição de *hardware* para ser sintetizado o FPGA. Somente assim, já que o FPGA pode utilizar 32 bits e ponto flutuante, vamos poder verificar com maior exatidão o ganho de eficiência e a corretude quando comparado à sua versão em *software.*

REFERÊNCIAS

[1] Y. k CHOI, J. CONG, Z. FANG, Y. HAO, G. REINMAN, and P. WEI. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. in 2016 53nd ACM/EDAC/IEEE Design Automation Conference, 2016, pp. 1–6.

[2] E. S. ALBUQUERQUE et al.. An FPGA-based accelerator for multiple real-time template matching, in 2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI), 2016, pp. 1–6.

[3] W. BERBERT, L. BERTINI, A. COPETTI. Aceleração de *hardware* em sistemas embarcados para aprendizado de máquina utilizando KNN em FPGA, XII Computer on the Beach, 7 a 9 de abril de 2021, Online, SC, Brasil in https://periodicos.univali.br/index.php/acotb/issue/view/619 Acesso em: 20 de agosto de 2023

[4] Wendell DINIZ, Vincent FREMONT, Isabelle FANTONI, and Euripedes NOBREGA. An fpga-based architecture for embedded systems performance acceleration applied to optimum-path forest classifier. Microprocessors and Microsystems, 52, 06 2017. doi: 10.1016/j.micpro.2017.06.013. Acesso em: 20 de agosto de 2023

[5] A. SHAWAHNA, S. M. SAIT and A. EL-MALEH. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. in IEEE Access, vol. 7, p. 7823-7859, 2019, doi: 10.1109/ACCESS.2018.2890150. Acesso em: 22 de agosto de 2023

[6] Yuwen SUN and Allen C. CHENG. Machine learning on-a-chip: A high-performance low-power reusable neuron architecture for artificial neural networks in ECG classifications. Computers in Biology and Medicine, 42(7):751 – 757, 2012. ISSN 0010-4825. doi: https://doi.org/10.1016/j.compbiomed.2012.04.007. URL http: //www.sciencedirect.com/science/article/pii/S001048251200073X. citado em: [3] Wanderson BERBERT, Luciano BERTINI, Alessandro COPETTI. Aceleração de *hardware* em sistemas embarcados para aprendizado de máquina utilizando KNN em FPGA, XII Computer on the Beach 7 a 9 de Abril de 2021, Online, SC, Brasil, p 400-407

[7] A. F. AQUINO NETO, S. R. FERNANDES. Acelerador do cálculo distância euclidiana em *Hardware*. Trabalho de Conclusão de Curso, 2020. p. 1-11.

[8] L. Andrade MACIEL, M. Alcântara SOUZA and H. Cota de FREITAS. Reconfigurable FPGA-Based *K-means*/ K-Modes Architecture for Network Intrusion Detection. in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, no. 8, pp. 1459-1463, Aug. 2020, doi: 10.1109/TCSII.2019.2939826. Acesso em: 22 de agosto de 2023

[9] L. Andrade MACIEL, M. Alcântara SOUZA, and H. Cota de FREITAS. Projeto e Avaliação de uma Arquitetura do Algoritmo de *Clusterização K-means* em VHDL e FPGA, in Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho, Campinas, 2017, pp. 256-267, doi: https://doi.org/10.5753/wscad.2017.254. Acesso em: 29 de agosto de 2023
[10] L. SILVA et al.. HPyC-FPGA - Integração de Aceleradores em FPGA de Alto Desempenho com Python para Jupyter Notebooks, in Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho, Florianópolis, 2022, pp. 193-204, doi: https://doi.org/10.5753/wscad.2022.226383. Acesso em: 29 de agosto de 2023
[11] L. BRAGANÇA et al.. An Open Source Custom *K-means* Generator for AWS Cloud FPGA Accelerators. in Anais do XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais, Evento Online, 2021, pp. 173-180.
[12] L. A. DIAS, J. C. FERREIRA and M. A. C. FERNANDES. Parallel Implementation of *K-means* Algorithm on FPGA. in IEEE Access, vol. 8, pp. 41071-41084, 2020, doi: 10.1109/ACCESS.2020.2976900. Acesso em: 5 de maio de 2023
[13] LOGISIM. Logisim (Versão 2.71). 2023. Universidade de Stanford. Disponível em: https://www.cburch.com/Logisim/. Acesso em: 27 de setembro de 2023.
[14] Elon Lages Lima, Espaços métricos, 6ªed, IMPA, Rio de Janeiro, 2020, p. 2, 3, 16 ISBN: 978-65-990528-7-3
[15] CARVALHO, João Bosco Pitombeira de, A raiz quadrada ao longo dos séculos, V Bienal da SBM – Sociedade Brasileira de Matemática, UFPB – Universidade Federal da Paraíba, 18 a 22 de outubro de 2010 – Disponível em: http://www.mat.ufpb.br/bienalsbm/arquivos/Mini_Cursos_Completos/MC11Completo.pdf Acesso em: 19 outubro 2023.
[16] MARS. Mips Assembly and Runtime Simulator (Versão 4.5). 2014. Pete Sanderson and Kenneth Vollmar. Disponível em: https://courses.missouristate.edu/kenvollmar/mars/download.htm Acesso em: 27 de setembro de 2023.

Apêndice A - A divergência do cálculo da distância entre o circuito e o SciKit-learn

Para exemplificar algumas das divergências apresentadas no nosso trabalho de conclusão de curso, tomamos com exemplo um indivíduo específico do conjunto de Indivíduos do *dataset* Iris (item 113), cujos valores, de suas características multiplicados por 10 (para torná-los valores inteiros) são: (57, 25, 50, 20) e calculamos a sua distância para os dois centroides mais próximos e divergentes, conforme a fórmula da distância euclidiana, como segue:

$$d(Q,\ K) = ((Q_1 - K_1)^2 + (Q_2 - K_2)^2 + \ \dots \ + (Q_n - K_n)^2)^{1/2} \qquad (1)$$

Onde Q_1, Q_2, ..., Q_n e K_1, K_2, ..., K_n são as coordenadas do indivíduo (Q) e do centroide (K).

O primeiro passo foi calcular as diferenças em módulo de cada uma das características do indivíduo para as de cada um dos centroides:

Indivíduo 113:	(57, 25, 50, 20)	Indivíduo 113:	(57, 25, 50, 20)
Centroide 2:	(64, 29, 43, 13)	Centroide 3:	(64, 27, 53, 19)
Diferença:	**(07, 04, 07, 07)**	**Diferença:**	**(07, 02, 03, 01)**

Em seguida elevamos os valores ao quadrado:

Indivíduo 113:	(57, 25, 50, 20)	Indivíduo 113:	(57, 25, 50, 20)
Centroide 2:	(64, 29, 43, 13)	Centroide 3:	(64, 27, 53, 19)
Diferença:	(07, 04, 07, 07)	Diferença:	(07, 02, 03, 01)
Quadrado:	**(49, 16, 49, 49)**	**Quadrado:**	**(49, 04, 09, 01)**

Depois, somamos os valores:

Indivíduo 113:	(57, 25, 50, 20)	Indivíduo 113:	(57, 25, 50, 20)
Centroide 2:	(64, 29, 43, 13)	Centroide 3:	(64, 27, 53, 19)
Diferença:	(07, 04, 07, 07)	Diferença:	(07, 02, 03, 01)
Quadrado:	(49, 16, 49, 49)	Quadrado:	(49, 04, 09, 01)
Soma:	**(163)**	**Soma:**	**(63)**

Por fim, extraímos a raiz quadrada:

Indivíduo 113:	(57, 25, 50, 20)	Indivíduo 113:	(57, 25, 50, 20)
Centroide 2:	(64, 29, 43, 13)	Centroide 3:	(64, 27, 53, 19)
Diferença:	(07, 04, 07, 07)	Diferença:	(07, 02, 03, 01)
Quadrado:	(49, 16, 49, 49)	Quadrado:	(49, 04, 09, 01)
Soma:	163	Soma:	63
SQRT:	**12,7671453**	**SQRT:**	**7,9372539**

Observe e compare com os resultados apresentados pelo circuito implementado e pelo Python:

Circuito:	**12**	**Circuito:**	**7**
Python:	**11,01394236**	**Python:**	**11,39246158**

Como se pode notar, a implementação em Python do *K-means* usando a biblioteca SciKit-learn[5] determinou que a menor distância é para o centroide 2, enquanto que os cálculos realizados pelo circuito demonstraram que o centroide 3 deveria ser o vencedor.

Como explicar essa divergência?

[5] https://colab.research.google.com/drive/1IcnGpnytbYIsPZEvhHieQmAk9lPmztp4#scrollTo=UK7yZtj3XGoQ