UNIVERSIDADE FEDERAL RURAL DO SEMIÁRIDO - UFERSA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
Trabalho de Conclusão de Curso (2024).

IMPLEMENTAÇÃO DE UM ACELERADOR EM HARDWARE PARA O CLASSIFICADOR DO ALGORITMO K-MEANS

Layne Paula Silveira[1], Sílvio Roberto Fernandes de Araújo[2]

[1]Graduanda – laynepsilveira@gmail.com
[2]Orientador – silvio@ufersa.edu.br

Resumo: A grande quantidade de dados gerados a cada minuto por centenas de aplicações utilizadas diariamente, torna cada vez mais custoso o trabalho de processá-los a um tempo tal que as informações resultantes sejam ainda úteis, assim surge a necessidade de soluções otimizadas que acelerem esses processos. O presente trabalho busca apresentar uma dessas soluções na implementação de um acelerador em hardware para o classificador do algoritmo K-means, utilizando a linguagem de descrição de hardware SystemVerilog. Utilizando alto grau de paralelismo e uma implementação de *pipeline* atingimos uma aceleração máxima de 35 vezes sobre a implementação sequencial em software e 3,38 vezes sobre a versão anterior da arquitetura do mesmo grupo de pesquisa.

Palavras-chave: K-means; Aceleração; Distância Euclidiana; FPGA.

1. INTRODUÇÃO

Os avanços tecnológicos resultaram no crescimento exponencial de dados gerados, armazenados e processados diariamente. Para o processamento, a análise e a utilização desses dados, é cada vez mais comum a utilização de algoritmos de agrupamento – ou *clustering* – para aprendizagem de máquina. O mais simples e popular entre eles é o algoritmo K-means, publicado inicialmente em 1955 [1]. Considerando apenas dados unidimensionais, o K-means tem uma complexidade de $O(qkt)$ [2], sendo q o número total de objetos (ou indivíduos), k o número de partições e t o número de iterações realizadas pelo algoritmo. Logo, o tempo de processamento para grandes quantidades de dados utilizando o algoritmo K-means pode ser proibitivo para muitas aplicações, considerando o uso de processadores de uso geral e soluções comuns em software. Ainda mais levando-se em conta que a maior parte dos dados gerados diariamente são multidimensionais.

A solução diametralmente oposta seria usar os processadores de propósito específico (ASIC - *Application-Specific Integrated Circuit*), projetados como solução de um único problema. Por sua própria definição e construção possuem o melhor desempenho, mas nenhuma flexibilidade, considerando mudanças de aplicação e o custo elevado para a produção.

Para superar isso, os FPGA (*Field Programmable Gate Array*) [3] se apresentam como a alternativa intermediária entre os processadores de propósito geral e os ASIC, tendo a capacidade de implementar circuitos específicos e paralelos, porém com a flexibilidade necessária a um mundo em constante movimento. Tal flexibilidade se dá pois os FPGA são constituídos de diversas LUTs (Look-Up Tables), que podem implementar qualquer lógica booleana, e roteadores que as interligam para expressões digitais mais complexas, como ilustrado pela Figura 1. As configurações das LUTs e roteadores são memorizadas em cada uma delas e podem ser alteradas em tempo de projeto. Logo, este projeto visa o desenvolvimento de um *hardware* acelerador do classificador para o K-means parametrizado utilizando a linguagem SystemVerilog que pode ser sintetizado em FPGA.

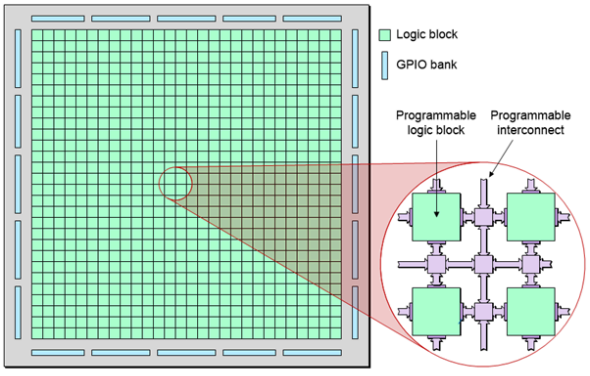


Figura 1. Visão geral e interna de um FPGA. (Lattice Semiconductor [4])

O restante do texto está organizado da seguinte forma: na seção 2 é apresentado o referencial teórico e uma revisão da literatura acerca do tema que serviram de base para a produção da pesquisa, na seção 3 são apresentados os detalhes da arquitetura proposta para o acelerador bem como sua implementação em *SystemVerilog*, na seção 4 são apresentados os resultados em comparação à implementação em software e à outras arquiteturas encontradas na literatura e, por fim, na seção 5 são apresentadas as conclusões e trabalhos futuros a partir desta pesquisa.

2. CONTEXTUALIZAÇÃO

2.1. Referencial Teórico

O objetivo do K-means é dividir um conjunto de dados com n indivíduos em k grupos ou *clusters*, cada um possuindo uma representação média dos indivíduos pertencentes, chamadas de centroides. O agrupamento é feito por similaridade – esta será calculada por uma métrica determinada.

O K-means é um algoritmo iterativo, ou seja, ele é executado de forma repetitiva em todos os dados, até que uma dada condição de parada seja atingida. O processo é ilustrado de maneira simplificada no Algoritmo 1. Os valores dos centroides são inicializados de forma aleatória antes da primeira iteração. A cada iteração, cada um dos indivíduos é comparado a todos os centroides para que possam ser atribuídos ao grupo do centroide que esteja mais próximo. Então os centroides são atualizados de acordo com os elementos atuais do grupo para que uma nova iteração aconteça. O critério de parada pode ser um número máximo de iterações ou um limite de mudanças nos centroides.

Algorithm 1 K-means

```
1.  procedure K-MEANS (k: clusters, D: conjunto de dados
2.      escolher k objetos de D como centroides dos clusters iniciais;
3.      repeat
4.          medir a distância do objeto d em relação a cada centroide;
5.          comparar todas as distâncias calculadas;
6.          (re)atribuir cada objeto d ao cluster de menor distância;
7.          atualizar o centroide do cluster;
8.      until clusters sem alterações ou até atingir o número de iterações;
9.      return conjunto de k clusters;
10. end procedure
```

Algoritmo 1. Pseudocódigo do K-means

A métrica de distância ou de similaridade é um ponto muito importante nas decisões de agrupamento, correspondendo à linha 4 do Algoritmo 1. Uma das métricas mais utilizadas, e que será utilizada neste projeto, é a distância euclidiana ao quadrado, devido à existência e unicidade do ponto que a minimiza, de acordo com a equação (1), onde c_j é o j-ésimo centroide, tal que $j = \{1, 2, \ldots, k\}$, sendo k a quantidade de *clusters* e $x \in X$, sendo $X = \{x_1, x_2, \ldots, x_q\}$ o conjunto com q dados, onde cada x_i e c_j possui n coordenadas.

$$f(c_j, x_i)^2 = \sqrt{\sum_{p=1}^{n} (c_{jp} - x_{ip})^2}, \; j = 1,\ldots, k; \; i = 1, \ldots, q. \tag{1}$$

O ponto que minimiza a distância de cada grupo é o centroide, descrito matematicamente como a média aritmética dos elementos do mesmo grupo [5], conforme a equação (2), onde q_j é a quantidade de dados agrupados no *cluster* j, ou seja, $q_j \leq q$.

$$c_j = \frac{x_1 + x_2 + \ldots + x_{q_j}}{q_j} \tag{2}$$

2.2. Trabalhos Relacionados

O crescente interesse em soluções para aceleração de algoritmos focados em processamento de grandes quantidades de dados para tarefas de aprendizagem de máquina, impulsionou a pesquisa e geração de trabalhos

nessa área.

A aplicação descrita em [6] propõe uma solução híbrida que calcula o algoritmo K-means e sua variação para dados categóricos, o algoritmo K-modes. Essa solução permite a configuração dos parâmetros de número de centroides e indivíduos, número de iterações e tipo de algoritmo em tempo de execução, suportando no máximo oito centroides por classificação. Porém, os dados são carregados em uma memória ROM em tempo de compilação, sendo assim necessária a recompilação do código de descrição de hardware como um todo em caso de alteração da base de dados. Embora a arquitetura receba palavras de 64 bits como entradas, elas têm diferentes significados dependendo do algoritmo. Para o K-means, cada palavra de 64 bits representa dois dados numéricos de 32 bits e, para o K-modes, cada palavra representa oito atributos categóricos de 8 bits. Essa implementação permite o processamento de duas informações por vez usando o K-means e oito informações por vez, utilizando o K-modes.

Já o trabalho [7] utiliza os ambientes PYNQ e Jupyter Notebook para criar uma solução híbrida hardware/software. Para a parte hardware, é utilizada aqui a síntese de alto nível (HLS – *High Level Synthesis* –, na sigla em inglês) que permite a descrição de hardware com a linguagem C++ por meio de diretivas de compilação. Com a escolha de usar HLS, o desenvolvimento é simplificado, porém se perde o controle sobre a construção do hardware que o projeto em nível de transferência de registradores (RTL – *Register Transfer Level* –, no inglês) permite. 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 [8] 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 pontos de dados e os centroides foi completamente paralelizado, mas todos os ajustes de parâmetros são feitos em tempo de compilação.

O trabalho de [9] 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 coordenadas dos G pontos de dados são processados junto com todas as coordenadas 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.

O presente trabalho busca contribuir com uma arquitetura com alto grau de paralelismo e de generalização, com a possibilidade de parametrização em tempo de execução, facilitando o processamento de grande quantidade de dados de diferentes modelos em menor tempo.

3. ARQUITETURA PROPOSTA

Esta seção apresentará a arquitetura do projeto e detalhes de implementação dos blocos SystemVerilog. O desenho da arquitetura parte de soluções anteriores [10], [11] desenvolvidas pelo mesmo grupo de pesquisa. A arquitetura leva em consideração paralelismo e economia de recursos de hardware, sempre priorizando o melhor desempenho temporal.

A Figura 2 mostra uma visão geral do projeto de arquitetura, formado por quatro blocos, cada um sendo responsável por uma das funções principais: gerenciamento de dados, cálculo e comparação de distâncias, atualização dos centroides e controle. O objetivo final dessa arquitetura é uma solução híbrida CPU-FPGA.

O primeiro bloco é responsável por lidar com a entrada e saída de dados por *streaming* e o armazenamento dos elementos do conjunto de dados que serão clusterizados, assim como os dados dos centroides. Neste trabalho chamamos os elementos como "indivíduos", e os valores de cada coordenada dos indivíduos como "características". Logo, a "Central de Dados" disponibiliza os dados para o restante da arquitetura bem como armazena os resultados calculados, os quais poderão ser exportados. Esse bloco também organiza os dados em indivíduos e centroides.

Nesta arquitetura cada dado será composto por até 32 características de até 32 bits. Os indivíduos poderão ser classificados em até 16 centroides. Esses quantitativos são os valores máximos nesta versão da arquitetura e definidos em tempo de projeto/compilação. Isso significa que tais valores definem a estrutura física do hardware e após a síntese não podem ser alterados, no entanto, a arquitetura possui entradas para definir logicamente a quantidade de clusters/centroides (entrada K) e de características (entrada N) em tempo de execução. O número de indivíduos será determinado pelo *streaming* de dados. Assim, se os valores desses parâmetros forem menores que o máximo, partes da arquitetura continuam existindo mas estarão desabilitadas.

O próximo bloco é o foco deste trabalho. Ele será responsável por calcular a distância entre cada indivíduo e cada centroide e classificar o indivíduo para o cluster mais próximo dele.

Em seguida o bloco responsável pela atualização dos centroides, recebe os indivíduos recém classificados e recalcula a média de cada cluster para serem usadas na próxima iteração do algoritmo. Este bloco está sendo desenvolvido em um trabalho paralelo.

Por último, o bloco de controle fica responsável por definir uma máquina de estados que gerencia todos os outros blocos e garante a sincronia da arquitetura como um todo. Em conjunto com a central de dados, esse bloco

será detalhado e implementado em trabalhos futuros.

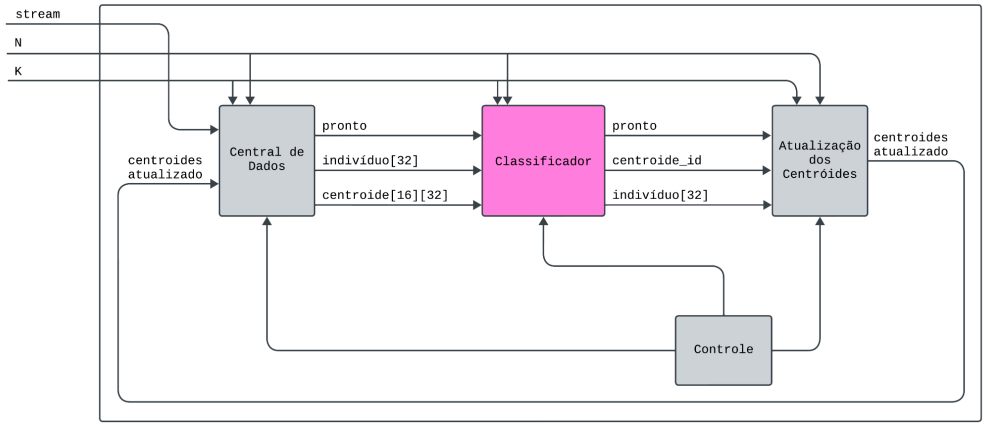


Figura 2. Visão geral da arquitetura completa do K-means desenvolvida pelo grupo de pesquisa. (Autoria própria)

3.1. Classificador

Este bloco recebe um indivíduo e todos os centroides por vez e é responsável por calcular todas as distâncias em paralelo e agrupar cada indivíduo em um cluster de acordo com a menor distância para os centroides. Além disso, ele também tem como entradas o número de características a serem consideradas em cada dado e o número de centroides a serem considerados para a classificação

Este trabalho teve por objetivo o desenvolvimento desse bloco, o qual será detalhado nesta seção e os resultados de simulação na seção seguinte. A Figura 3 mostra a interface da sua implementação em *SystemVerilog* e a Figura 4 exemplifica sua arquitetura interna.

```
4  module classificador
5  #(
6      DATA_WIDTH = 32,
7      N_CARAC = 32,
8      N_CENTR_K = 16
9  )
10 (
11     input logic clock, reset, active,
12     input logic [7:0] caract_num, centr_num,
13     input logic [DATA_WIDTH-1:0] individual_in [0:N_CARAC-1],
14     input logic [DATA_WIDTH-1:0] centroides [0:N_CENTR_K-1][0:N_CARAC-1],
15
16     output logic [DATA_WIDTH-1:0] individual_out [0:N_CARAC-1],
17     output logic [DATA_WIDTH-1:0] distance,
18     output logic [3:0] centroide_id,
19     output logic valid
20 );
```

Figura 3. Interface em *SystemVerilog* do bloco classificador. (Autoria própria)

Além disso, tem como parâmetros $DATA\_WIDTH$, N_CARAC e N_CENTR_K, que representam respectivamente a largura máxima em bits de cada característica, o número máximo de características e o número máximo de centroides. Os parâmetros de um módulo *SystemVerilog* são definidos em tempo de compilação e definem o que vai ser sintetizado no circuito final, o que significa, nesse caso, que fisicamente nós teremos suporte a dados de até 32 características, cada uma com até 32 bits e poderemos classificar os dados em até 16 centroides.

Para possibilitar a parametrização do cálculo em tempo de execução, usaremos as entradas *caract_num* e *centr_num*, representados na Figura 4 por *N* e *K*, respectivamente, que vão definir quantas dessas estruturas físicas serão usadas a cada iteração.

O projeto e a implementação foram feitos visando o máximo de paralelismo e a aplicação de *pipeline* no processamento dos dados. No total, o classificador leva 35 ciclos para retornar o primeiro resultado e depois disso apresenta um resultado por ciclo.

Como ilustrado na Figura 4, para realizar o cálculo das distâncias entre um indivíduo e todos os centroides em paralelo, usaremos 16 instâncias do bloco de operações, que será detalhado na seção 3.3. Cada uma dessas instâncias recebe um centroide e uma cópia do indivíduo atual, além do número de características *N*. Ao final do cálculo, todas as instâncias seguem para a árvore de comparadores, especificada na seção 3.4 que retorna a distância do indivíduo ao centroide mais próximo e o identificador desse centroide.

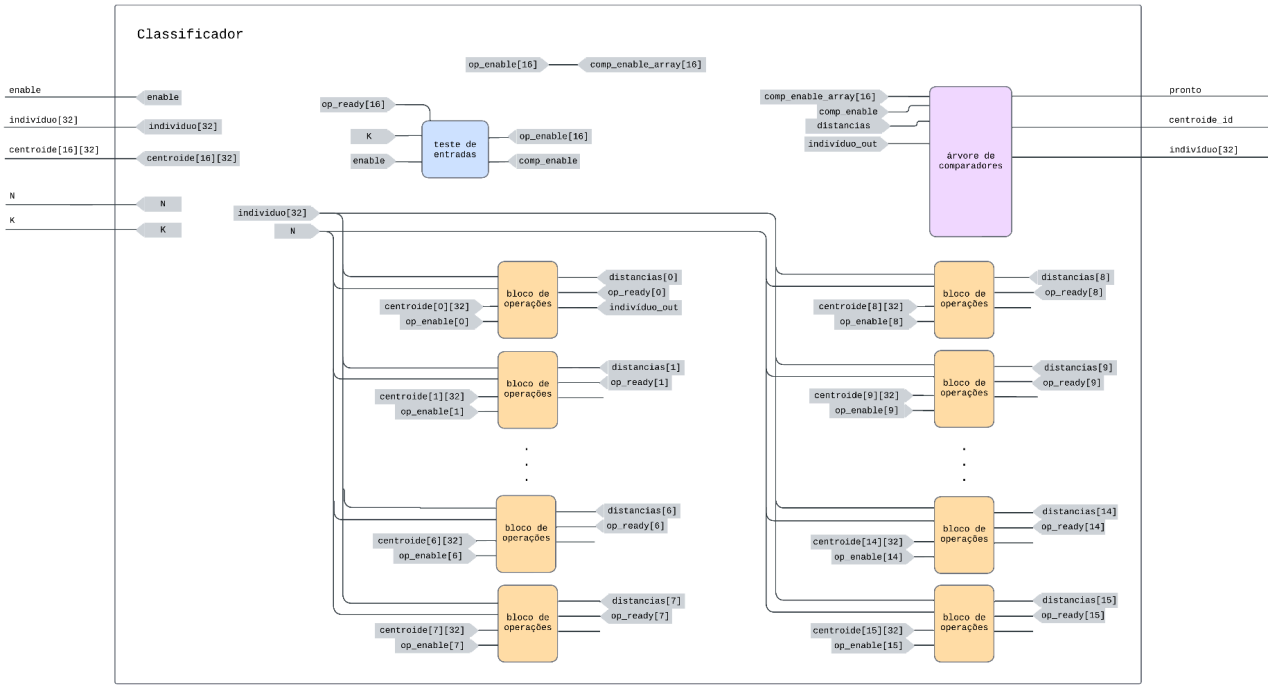


Figura 4. Ilustração da arquitetura interna do bloco classificador. (Autoria própria)

3.2. Teste de entradas

O teste de entradas garante a parametrização em tempo de execução, codificando o número de centroides em um vetor de sinais de *enable*, a ser distribuído às instâncias de cálculo e definir quais valores serão considerados para o resultado final. Além de garantir a sincronia entre os blocos de operações e a árvore de comparadores. A Figura 5 representa esse bloco e suas entradas.

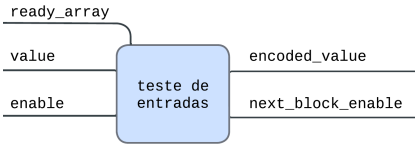


Figura 5. Teste de entradas. (Autoria própria)

A implementação é bastante simples, como demonstra a Figura 6, o valor de entrada $value$ é traduzido em um número de $OUTPUT\_WIDTH$ bits com os primeiros $value$ bits ativos. Por exemplo, com $value = 4$ e $OUTPUT\_WIDTH = 8$, teremos como saída os 4 últimos bits, dos 8, setados em 1 e os demais em zero, ou seja, a saída seria assim: 00001111. No bloco classificador, o teste de entradas está parametrizado com $OUTPUT\_WIDTH = 16$.

Depois disso, teremos a comparação de $output\_done$ – entrada que representa o conjunto de sinais de pronto dos blocos de operação – e $current\_output$ – saída atribuída aos sinais de *enable* dos blocos de operação –, o que vai definir que, quando todos os blocos de operação ativos estiverem prontos, a próxima etapa, nesse caso a árvore de comparadores, pode começar sua execução, recebendo o mesmo conjunto de sinais de *enable*.

```
4  module input_tester #
5    (
6      parameter INPUT_WIDTH = 8,
7      parameter OUTPUT_WIDTH = 32
8    )
9    (
10     input  logic enable,
11     input  logic [INPUT_WIDTH-1:0] value, // número a ser codificado
12     input  logic [OUTPUT_WIDTH-1:0] ready_array, // ready array
13     output logic [OUTPUT_WIDTH-1:0] encoded_value, // enable array
14     output logic next_block_enable
15   );
16     logic [OUTPUT_WIDTH-1:0] mux_input [1:0], input_encoded, encoded_output;
17     assign encoded_value = encoded_output;
18     encoder #(.ENCODE_WIDTH(INPUT_WIDTH), .DATA_WIDTH(OUTPUT_WIDTH)) dut0(
19         .active_input(value),
20         .active_output(input_encoded)
21     );
22     assign mux_input[0] = 1'b0;
23     assign mux_input[1] = input_encoded;
24     mux2_1 #(.C_AXIS_TDATA_WIDTH(OUTPUT_WIDTH)) mux0(
25         .data(mux_input),
26         .select(enable),
27         .mux_output(encoded_output)
28     );
29     comparator #(.DATA_WIDTH(OUTPUT_WIDTH)) comp0(
30         .x(input_encoded),
31         .y(ready_array),
32         .is_equal(next_block_enable)
33     );
34 endmodule
```

Figura 6. Teste de entradas - *SystemVerilog*. (Autoria própria)

O bloco de operações também usa uma instância do teste de entradas parametrizado com $OUTPUT\_WIDTH = 32$.

3.3. Bloco OP

O próximo bloco que vamos detalhar é o módulo de operações. Esse é o bloco responsável por calcular a distância euclidiana ao quadrado entre um indivíduo e um centroide, considerando todas as características de cada um, definida na equação (3). Recebe como entrada todas as características – no máximo 32 – de um indivíduo e de um centroide, assim como o número de características que devem ser consideradas em cada iteração.

(3)

$$f(c_j, x_i) = \sqrt{\sum_{p=1}^{n} (c_{jp} - x_{ip})^2}, \; j = 1,\ldots, k; \; i = 1, \ldots, q.$$

Para o cálculo das distâncias de cada par de características, teremos 32 instâncias do módulo de distância quadrada, especificado na seção 3.3.1, que recebem os sinais de enable do módulo de teste de entradas e executam em paralelo. Os resultados de todos esses blocos, bem como cada característica, são então enviados à árvore de somas, detalhada na seção 3.3.2, que por sua vez envia seu resultado para o módulo de raiz quadrada – explicado na seção 3.3.3 –, continuando também a repassar os valores do indivíduo, como pode ser visto na Figura 7.

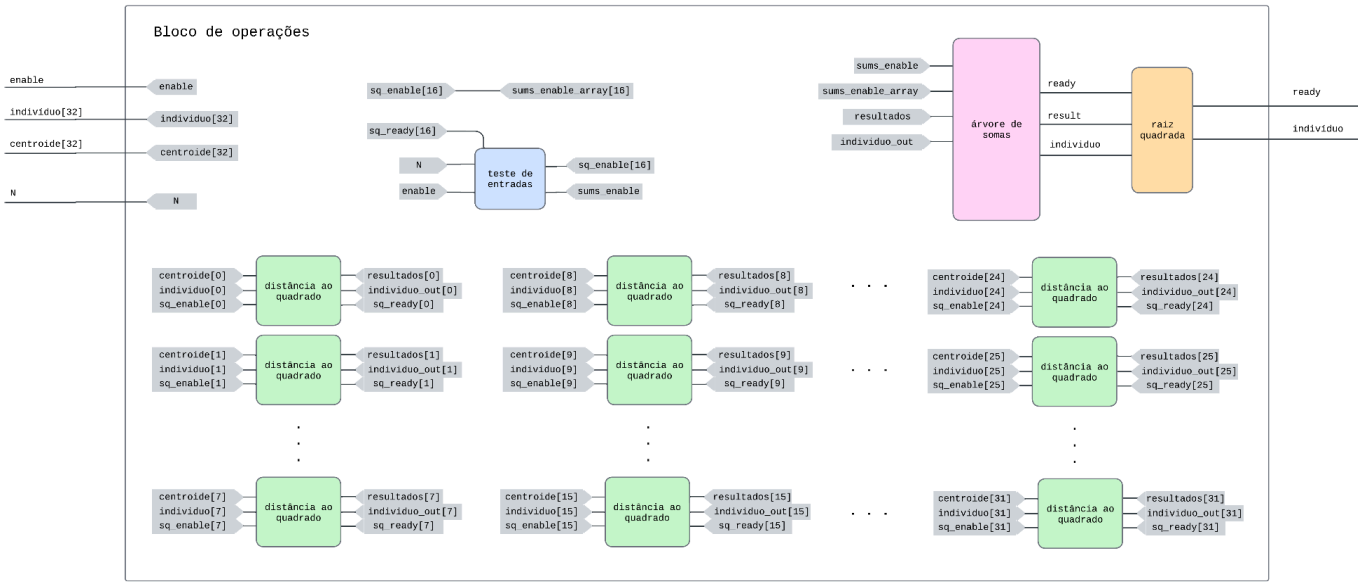


Figura 7. Bloco OP. (Autoria própria)

3.3.1. Cálculo da distância ao quadrado

Esse bloco é o responsável pela parte do cálculo especificado na equação (3), que define a distância entre uma característica do centroide c_j à característica correspondente do indivíduo onde x_i: $(c_{jp} - x_{ip})^2$. A Figura 8 ilustra sua arquitetura interna.

Aqui começamos a detalhar os estágios de *pipeline* da arquitetura, tanto para o cálculo quanto para o valor da característica do indivíduo. No total esse bloco leva 3 ciclos para devolver o primeiro resultado. Na Figura 8 são ilustrados 3 "colunas" de registradores, de modo que aqueles que estão na mesma coluna estão trabalhando em paralelo. Os dados fluem da esquerda para direita, passando pelas 3 colunas de registradores, os quais são atualizados a cada ciclo. Esta é a forma que a técnica de *pipeline* é implementada, permitindo acelerar um conjunto de trabalho. Isso quer dizer que para que o primeiro dado que chega a este módulo tenha seu resultado computado na saída são necessários 3 ciclos, correspondente aos 3 registradores no caminho, mas no quarto ciclo um novo resultado está pronto e assim por diante a cada ciclo. O paralelismo está entre as fases do *pipeline* em que cada dado se encontra, e neste caso como são 3, essa é a aceleração máxima que pode ser obtida em relação a uma solução sem esta técnica.

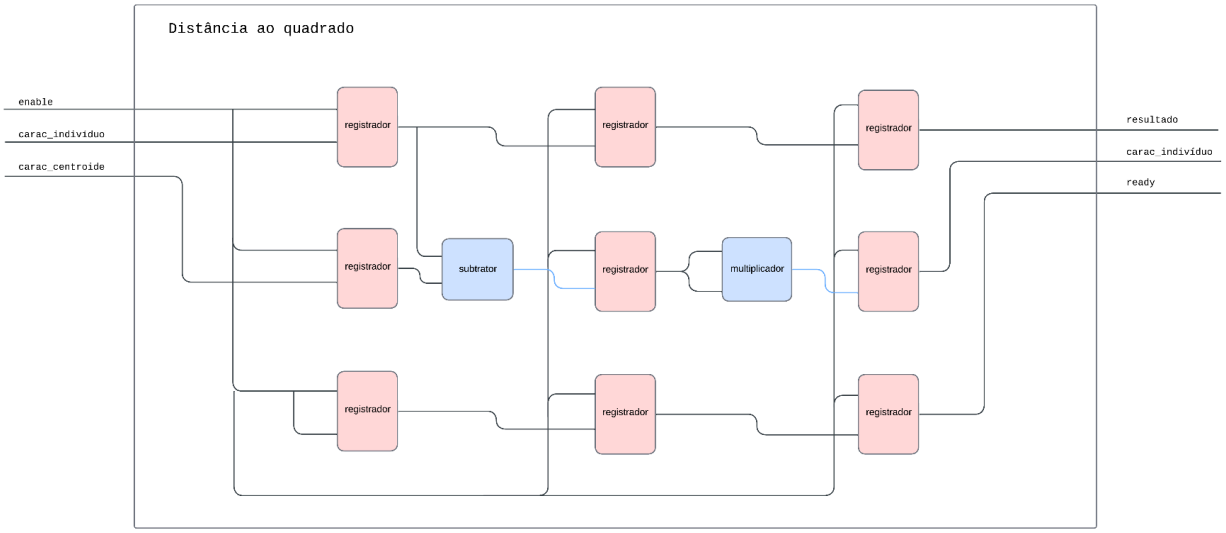


Figura 8. Bloco Square Distance. (Autoria própria)

As entradas desse bloco são dados de 32 bits, mas as saídas após a multiplicação, serão dados de 64 bits, como podemos ver na interface do módulo *SystemVerilog* na Figura 9.

```
4  module square_distance #
5      (
6          parameter DATA_WIDTH = 32,
7          parameter PIPE_NUM = 3
8      )
9      (
10         input logic clock, reset, active,
11         input logic [DATA_WIDTH-1:0] indiv_in,
12         input logic [DATA_WIDTH-1:0] centr_in,
13         output logic [DATA_WIDTH-1:0] indiv_out,
14         output logic [(2*DATA_WIDTH)-1:0] square_result,
15         output logic ready
16     );
```

Figura 9. Bloco Square Distance - Interface. (Autoria própria)

3.3.2. Árvore de somas

A árvore de somas implementa a fase do somatório da equação (3). Recebendo até 32 valores de entrada, os quais são somados aos pares em paralelo, e os resultados intermediários também somados aos pares, definindo uma árvore com profundidade de 5 níveis, os quais também aplicam a técnica de *pipeline*. A Figura 10 ilustra a arquitetura interna do bloco "Sum 4".

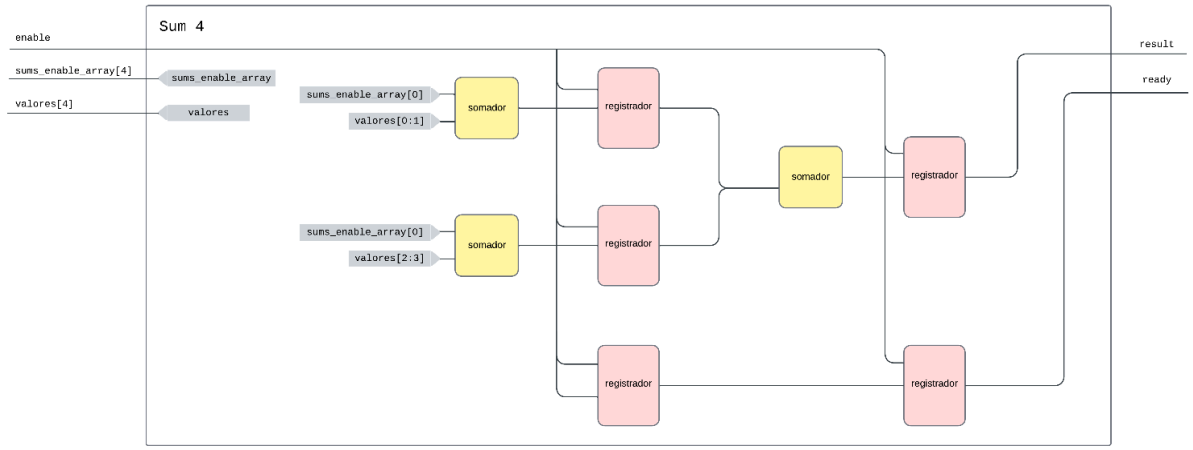


Figura 10. Sums 4. (Autoria própria)

A árvore de somas, ilustrada na Figura 11, foi hierarquizada. Assim, definimos um módulo chamado “Sum 4” que implementa 2 níveis da árvore de somas, ou seja, recebe 4 valores, que são somados como 2 pares em paralelo, e os resultado vão para um próximo somador. O módulo “Sum 4” implementa um pipeline que leva 2 ciclos para entregar o primeiro resultado. Seguindo essa lógica, definimos o módulo “Sum 8” que instancia o bloco “Sum 4” duas vezes, levando até 3 ciclos para a soma de 8 valores.

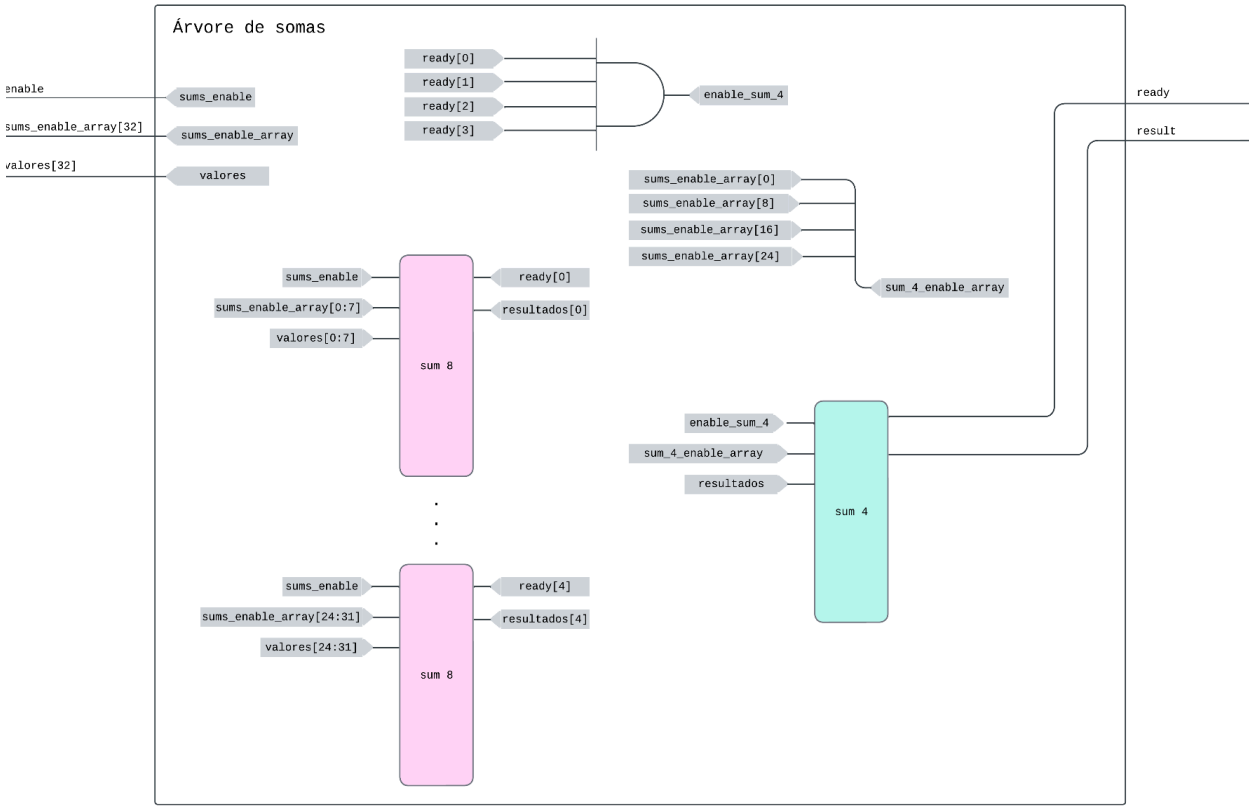


Figura 11. Árvore de Somas. (Autoria própria)

A árvore de somas utiliza uma instância do bloco “Sums 4” e quatro instâncias em paralelo do bloco “Sums 8”, que por sua vez, utiliza duas instâncias em paralelo do bloco “Sums 4” e mais um somador na última camada. Assim, o pipeline completo da árvore de somas gasta no máximo 5 ciclos.

As características dos indivíduos também são aplicadas a um *pipeline* de mesma profundidade, para que ao final do processamento o resultado da classificação seja acompanhado pelo indivíduo completo.

A Figura 12 mostra a implementação da esteira dos indivíduos dentro do módulo da árvore de somas. Os blocos da árvore de comparadores e raiz quadrada usam a mesma técnica no seu próprio *pipeline* de indivíduos e por isso não serão demonstrados aqui.

```
29      // registrador de deslocamento - define os estágio ativos do pipeline
30      logic [PIPE_DEPTH-1:0]    reg_active;
31      pipeline #(.DATA_WIDTH(DATA_WIDTH), .PIPE_NUM(PIPE_DEPTH)) pipe_reg_active (.clock(clock),
    .reset(reset), .active(active), .ready(reg_active));
32      // definição de fios para serialização
33      logic [DATA_WIDTH-1:0] inner_ind_data[0:PIPE_DEPTH-1][0:31];
34      assign ind_data_out = inner_ind_data[PIPE_DEPTH-1];
35      //conjuntos de registradores para cada estágio
36      cluster_register #(.DATA_WIDTH(DATA_WIDTH), .DATA_DEPTH(32)) cluster_register_0 (.clock(clock),
    .reset(reset), .active(active), .register_input(ind_data_in), .register_output(inner_ind_data[0]));
37      genvar i;
38      generate
39          for (i = 1; i < PIPE_DEPTH; i++) begin : reg_gen_block
40              cluster_register #(.DATA_WIDTH(DATA_WIDTH), .DATA_DEPTH(32)) cluster_register_gen
    (.clock(clock), .reset(reset), .active(reg_active[i-1]), .register_input(inner_ind_data[i-1]),
    .register_output(inner_ind_data[i]));
41          end
42      endgenerate
```

Figura 12. Pipeline dos indivíduos na árvore de somas. (Autoria própria)

3.3.3. Raiz quadrada

O cálculo é feito usando recursos de cálculo numérico, baseado no método de Heron [12], demonstrado no Algoritmo 2. Sendo "S" o número a ser extraída a raiz quadrada e "*n*" a quantidade de iterações. A precisão do resultado é proporcional ao valor de "*n*".

Algorithm 2 Raiz quadrada

Entrada: S, n

Saída: $\sqrt{S}$

1. **início**
2. $Y = 1 + (\frac{1}{2}S)$;
3. **para** (int i = 0; i < n; i++) **faça**
4. $X_i = \frac{1}{2}(Y + \frac{S}{Y})$;
5. $Y = X_i$;
6. **fim**
7. **fim**

Algoritmo 2. Pseudocódigo da raiz quadrada

Para implementação em hardware, cada repetição consiste em um estágio de pipeline. Por meio de experimentação o valor de *n* do Algoritmo 2 que definiu uma boa precisão numérica para dados de 64 bits foi 20, sendo esta, portanto, a quantidade de estágios do *pipeline* do *hardware* da raíz quadrada. A raíz quadrada é realizada sobre as diferenças elevadas ao quadrado, conforme equação (1). Como em hardware os valores de 32 bits elevados ao quadrado resultam em valores de 64 bits, a entrada desse bloco de hardware é de 64 bits e saída de 32 bits.

A Figura 13 mostra o bloco que representa um dos estágios do *pipeline* enquanto a Figura 14 mostra a instanciação parametrizada desse bloco no módulo de raiz quadrada.

```
module sqrt_unit #
(
    parameter DATA_WIDTH = 32
)
(
    input logic clock, reset, active,

    input  logic [DATA_WIDTH-1:0] init_value_in,    // entrada valor inicial
    input  logic [DATA_WIDTH-1:0] current_sqrt,     // valor calculado pela unidade anterior

    output logic [DATA_WIDTH-1:0] init_value_out,   // saida valor inicial
    output logic [DATA_WIDTH-1:0] next_sqrt         // valor calculado por esta unidade
);

    logic [DATA_WIDTH-1:0] X;
    logic [DATA_WIDTH-1:0] Y;

    assign init_value_out = X;
    assign next_sqrt = Y;

    always @(posedge clock, posedge reset) begin
        if (reset) begin
            X = 1'b0;
            Y = 1'b0;
        end
        else if (active) begin
            X = init_value_in;
            Y = (current_sqrt + X/current_sqrt) / 2;
        end
        else begin
            X = X;
            Y = Y;
        end
    end
endmodule
```

Figura 13. Bloco sqrt_unit. (Autoria própria)

A instanciação utiliza o bloco *generate*. Essa função do *SystemVerilog* permite a definição de diversas instâncias de uma forma simplificada. O laço de repetição descrito não será sintetizado na placa, mas desenrolado em instâncias separadas.

```
    genvar i;
    generate
        for (i = 1; i < PIPE_NUM; i++) begin
          sqrt_unit #(.DATA_WIDTH(DATA_WIDTH*2)) sq(.clock(clock), .reset(reset),
.active(unit_active[i-1]), .init_value_in(value[i-1]),
.current_sqrt(sqrt_unit_result[i-1]), .init_value_out(value[i]),
.next_sqrt(sqrt_unit_result[i]));
        end
    endgenerate
```

Figura 14. Instanciação dos blocos sqrt_unit. (Autoria própria)

3.4. Árvore de comparadores

A árvore de comparadores recebe os valores dos resultados dos blocos de operação e calcula o menor valor, atrelando-o ao identificador do centroide e aos valores das características dos indivíduos.

A árvore de comparadores também foi hierarquizada, assim como a de somadores. Logo, definimos um módulo chamado “comp_with_id” que implementa um nível da árvore, ou seja, recebe 2 valores. A Figura 15 ilustra esse módulo, que leva um ciclo para devolver um resultado.

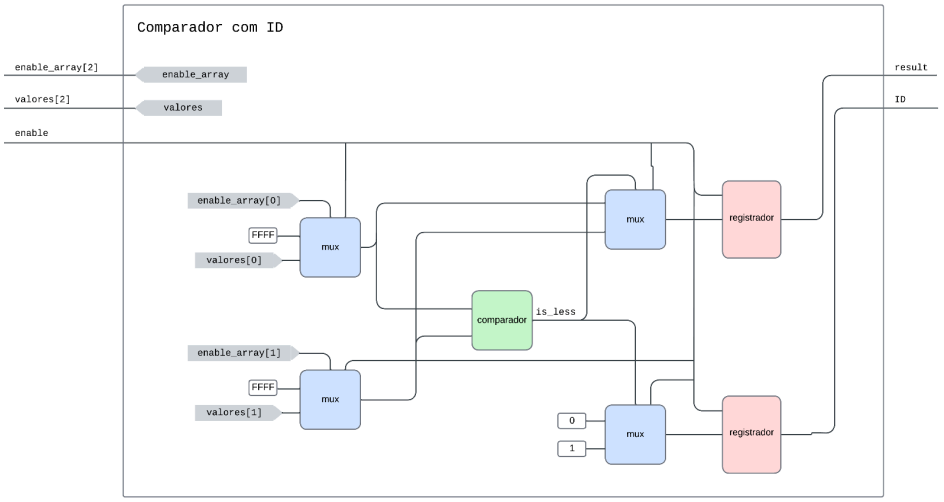


Figura 15. Bloco comp_with_id. (Autoria própria)

O módulo “Comp 4” gasta 3 ciclos de clock para retornar o primeiro resultado, utilizando duas instâncias “comp_with_id” em paralelo e uma serial na última camada. A árvore de comparadores, cuja arquitetura interna está ilustrada na Figura 16, utiliza a mesma lógica, instanciando o módulo “Comp 4” cinco vezes, gastando um total de 7 ciclos para o primeiro resultado.

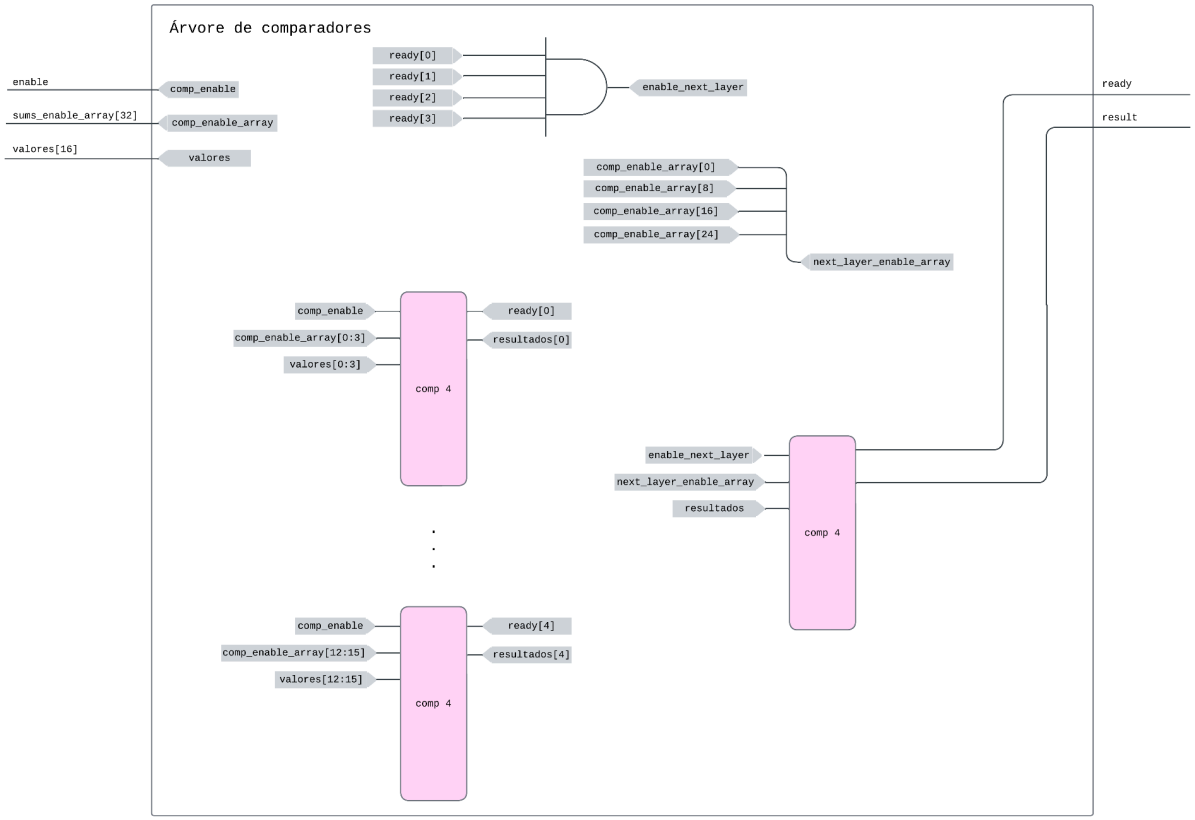


Figura 16. Árvore de comparadores. (Autoria própria)

4. RESULTADOS

Os resultados foram gerados utilizando o modelo de simulação funcional e não temporal, o que significa que não foi levado em conta o tempo real em segundos para a execução, sendo calculado o tempo apenas em função do número de ciclos. As simulações foram feitas utilizando *testbenches* na linguagem *SystemVerilog* aliado a uma versão para comparação em software, desenvolvida em *Python*, que fornece os dados de entrada e resultados esperados. Assim, cada resultado calculado pela implementação em *SystemVerilog* é comparado aos resultados esperados através de arquivos.

Os testes da arquitetura foram realizados no ambiente de desenvolvimento EdaPlayground [13], utilizando o compilador Aldec Riviera Pro 2022.04. Devido às limitações de memória e tempo de execução do ambiente, os

testes foram reduzidos a um máximo de 160 indivíduos, o que já ilustra o funcionamento da arquitetura, seu paralelismo e potencial de aceleração.

4.1. Análise de paralelismo e *pipeline*

A arquitetura foi construída visando a maior aceleração possível, para isso exploramos o paralelismo e a técnica de *pipeline*.

O primeiro indivíduo a ser processado precisa percorrer a arquitetura completa, contando somente com a aceleração decorrente do paralelismo da arquitetura, tendo um resultado em $T = 35$, sendo esse o número de ciclos total da arquitetura (vide Tabela 1). A partir do segundo indivíduo podemos ver a aceleração possibilitada pelo *pipeline* e teremos um novo resultado sendo gerado a cada ciclo, como podemos ver na forma de onda da Figura 17, onde o sinal "clock_count" mostra a contagem de ciclos da simulação.

Tabela 1. Quantidade de ciclos por módulo

Circuito	Ciclos
Distância ao quadrado	3
Árvore de somas	5
Raiz quadrada	20
Árvore de comparadores	7
Total	35

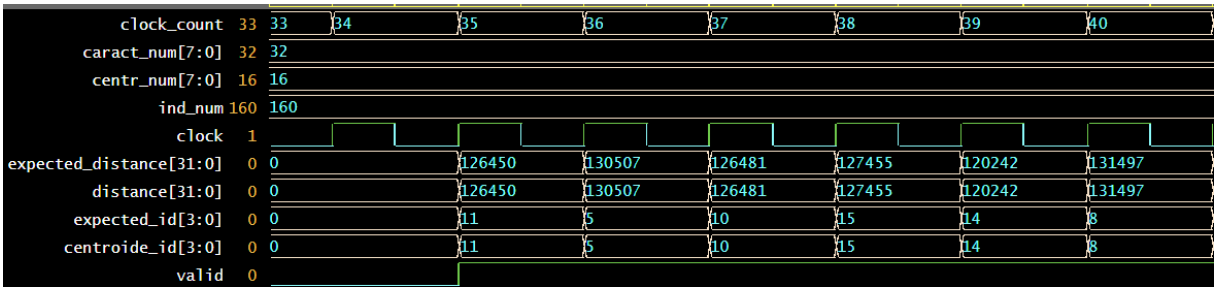


Figura 17. Forma de onda. (Autoria própria)

A Tabela 2 mostra os resultados, com variações de todos os parâmetros, sendo N o número de características por dado, K o número de centroides e Q o número de indivíduos. A coluna "Primeiro resultado" indica quantidade de ciclos para arquitetura calcular a distância e classificar o primeiro indivíduo, seguido pela quantidade de ciclos com *pipeline* e sem *pipeline*, e a aceleração da versão com *pipeline* em relação a sem *pipeline*.

Em uma implementação sem o uso de *pipeline*, o processamento dos dados acontece sequencialmente. Tendo em conta que a arquitetura realiza o processamento de 1 indivíduo em 35 ciclos, consideramos que uma execução sequencial gastaria um tempo $T = 35 \times Q$. Utilizamos essa equação para gerar os dados da coluna "#Ciclos sem pipeline", onde podemos ver uma aceleração de até 28 vezes em comparação a uma execução sem *pipeline* para este conjunto de testes.

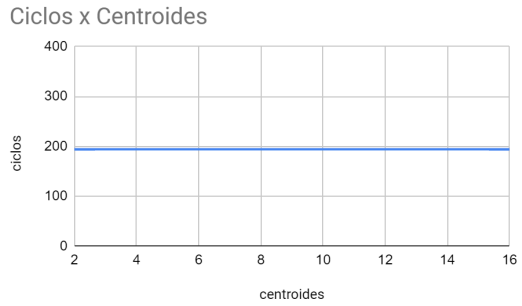
Tabela 2. Valores simulados no EdaPlayground

Teste	N	K	Q	Primeiro resultado	#Ciclos com pipeline	#Ciclos sem pipeline	Aceleração
1	4	2	40	35	74	1400	18
2	4	4	40	35	74	1400	18
3	4	8	40	35	74	1400	18
4	4	16	40	35	74	1400	18

5	8	2	80	35	114	2800	24
6	8	4	80	35	114	2800	24
7	8	8	80	35	114	2800	24
8	8	16	80	35	114	2800	24
9	16	2	120	35	154	4200	27
10	16	4	120	35	154	4200	27
11	16	8	120	35	154	4200	27
12	16	16	120	35	154	4200	27
13	32	2	160	35	194	5600	28
14	32	4	160	35	194	5600	28
15	32	8	160	35	194	5600	28
16	32	16	160	35	194	5600	28

A partir desses dados, fixando-se os valores de N em 32 e Q em 160, e variando-se os valores de K, temos que o número de centroides não interfere no tempo de execução do circuito, como ilustrado pelo gráfico da Figura 18.a. Da mesma forma na Figura 18.b, temos a demonstração de que o número de características também não interfere no tempo de execução do circuito. Esses dois gráficos mostram a eficiência da paralelização da arquitetura, visto que a medida de similaridade acontece sob todas as características de um mesmo indivíduo com todos os centroides em paralelo.

a)



b)

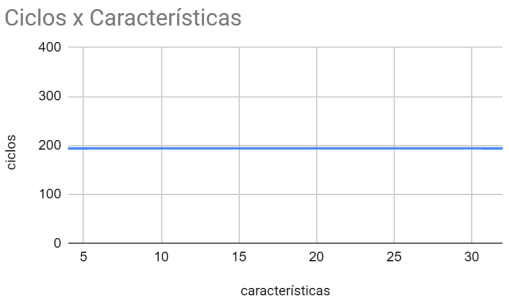


Figura 18. Relação entre a quantidade de características e centroides com o tempo total do circuito. (Autoria própria)

O gráfico da Figura 19 mostra que, fixando-se N em 32 e K em 16, o tempo do circuito varia linearmente com a variação de Q. Assim, o tempo total T de execução pode ser descrito em função do número de ciclos necessários para o cálculo do primeiro resultado e do número de indivíduos Q, de acordo com equação a seguir:

(4)

$$T = 35 + (Q - 1)$$

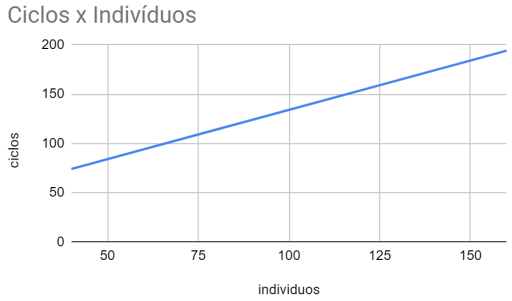


Figura 19. Relação entre a quantidade de indivíduos com o tempo total do circuito. (Autoria própria)

4.2. Aceleração

Para a avaliação da aceleração da arquitetura implementada, apresentamos primeiro a aceleração máxima possível. Considerando que o número de indivíduos Q será armazenado em um registrador de 32 bits e interpretado como um número inteiro sem sinal, temos que $Q_{max} = 2^{32} = 4.294.967.296$. Através da equação $T = 35 + (Q - 1)$ definida na seção anterior, podemos calcular o tempo total da arquitetura em ciclos para este valor de Q. Utilizando a mesma função utilizada na seção anterior calculamos também o tempo que seria necessário para a execução sem *pipeline*. Assim, chegamos à conclusão que a aceleração máxima possível para o circuito é de 34,99 vezes. A tabela a seguir condensa esses resultados.

Tabela 3. Aceleração máxima.

Q máximo	Ciclos sem pipeline	Ciclos com pipeline	Aceleração
4.294.967.296	150.323.855.360	4.294.967.330	34,99

Para comparações com uma solução em software, utilizamos a implementação em assembly do processador MIPS apresentada em [11], para que se pudesse usar o número de instruções assembly como métrica de comparação ao circuito implementado neste trabalho.

A implementação assembly possui algumas instruções que vão executar apenas uma vez e outras instruções que estão dentro de um laço de repetição que dependem dos parâmetros do algoritmo. Assim, para simplificar, foram consideradas apenas as 34 instruções dentro do laço de repetição. Para simplificar as comparações vamos considerar que todas as instruções executam em 1 único ciclo e que o processador implementa um *pipeline* de 5 estágios.

Em todos os testes realizados nesta etapa utilizou-se de dados genéricos com 4 características e clusterização em 4 grupos. A quantidade de indivíduos variou de 4 a 65536, conforme a Tabela 4. Nesta tabela pode-se ver a quantidade de ciclos no MIPS com pipeline de 5 estágios, do projeto Logisim apresentado em [11] e da arquitettura proposta neste trabalho. As últimas duas colunas são a comparação do quanto a proposta atual acelerou em relação ao MIPS e ao projeto Logisim. Nestas comparações a arquitetura proposta neste trabalho em relação ao MIPS com *pipeline*, pode chegar até pouco mais de 33 vezes mais rápida. Uma constatação também é que essa aceleração cresce com o número de indivíduos, graças a seu paralelismo, com uma tendência de estabilização a partir de 2048 indivíduos. Em relação ao projeto Logisim, a estabilização da aceleração acontece já a partir de 1024 indivíduos, atingindo seu máximo em 4 vezes.

Tabela 4. Comparativo com a quantidade de instruções MIPS

Indivíduos	Ciclos MIPS com pipeline	Ciclos Projeto em Logisim	Ciclos Arquitetura Implementada	Aceleração em relação ao MIPS	Aceleração em relação ao proj. Logisim
4	167	53	38	4,39	1,39
8	303	69	42	7,21	1,64
16	575	101	50	11,5	2,02
32	1119	165	66	16,95	2,50
64	2207	293	98	22,52	2,99

128	4383	549	162	27,05	3,39
256	8735	1061	290	30,12	3,66
512	17439	2085	546	31,94	3,82
1024	34847	4133	1058	32,94	3,91
2048	69663	8229	2082	33,46	3,95
4096	139295	16421	4130	33,73	3,98
8192	278559	32805	8226	33,86	3,99
16384	557087	65573	16418	33,93	3,99
32768	1114143	131109	32802	33,97	4,00
65536	2228255	262181	65570	33,98	4,00

A Tabela 5 mostra a comparação dos resultados deste trabalho a trabalhos anteriores do mesmo grupo de pesquisa. Apresentamos uma aceleração no tempo para o primeiro resultado de 1,65 com relação a [10] e de 1,05 com relação a [11]. Para o tempo de processamento de cada novo indivíduo, temos uma aceleração de 10 e 4 vezes, em relação à [10] e [11], respectivamente. Por fim, para processamento de 128 indivíduos, a arquitetura implementada apresenta uma aceleração de 8,19 vezes sobre [10] e de 3,38 vezes sobre [11].

Tabela 5. Comparação a trabalhos anteriores.

Arquitetura	**Ciclos para o primeiro resultado**	**Ciclos para cada novo indivíduo**	**Ciclos para processamento de 128 indivíduos**
NETO, 2020	58	10	1328
GUIMARÃES, 2023	37	4	549
Presente Trabalho	35	1	162

4.3. MAPEAMENTO PARA FPGA

Para a implementação de um algoritmo em linguagem de descrição de hardware para uma placa FPGA precisamos passar por duas fases: síntese e implementação. A primeira fase mapeia todos os recursos da placa que serão necessários para a execução do algoritmo e a segunda fase faz a instalação desse mapeamento para a placa alvo.

A placa disponível é a Ultra96-V1 que possui FPGA da Xilinx, porém a síntese mostrou que a arquitetura proposta no presente trabalho exige mais recursos do que essa placa pode oferecer. O gráfico da Figura 20 mostra a relação entre os recursos necessários e os recursos disponíveis na placa. Dessa forma, não foi possível realizar os testes de integração do acelerador em FPGA com o software.

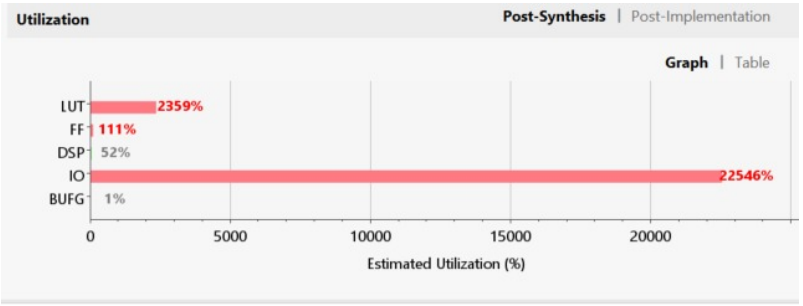


Figura 20. Relação entre os recursos necessários e os recursos disponíveis na placa Ultra96-V1. (Autoria própria)

5. CONCLUSÕES

Este trabalho apresenta a implementação de um acelerador em hardware para o classificador do algoritmo K-means. Os ganhos de desempenho se devem à completa paralelização do cálculo de distância e à implementação de *pipeline* através de toda a arquitetura.

Os resultados de simulação demonstram que o cálculo de similaridade sob todas as características de um dado indivíduo em relação a todos os centroides em paralelo, somado ao *pipeline*, representam um alto desempenho, de modo que esses parâmetros não influenciam o tempo de execução. O tempo de resposta de tal arquitetura está relacionado apenas a quantidade de indivíduos, sendo a latência mínima do pipeline de 35 ciclos.

Dessa forma, a arquitetura proposta apresentou um potencial de aceleração de até 33 vezes em relação a uma implementação em *software* em um processador MIPS com *pipeline* e melhoria de 8 e 3 vezes em relação a versões anteriores dessa arquitetura. A projeção para a aceleração máxima em comparação a uma implementação sem *pipeline* é de 35 vezes.

Trabalhos futuros incluem a integração desse classificador com a central de dados e o atualizador de centroides, conforme a arquitetura completa do K-means apresentada no início deste texto e realizar simulações temporais, considerando o tempo real de execução do circuito. Além disso, para a solução de computação heterogênea CPU-FPGA será necessário realizar a adaptação das interfaces de entrada/saída da arquitetura geral para o padrão AXI, assim, poderemos avaliar o desempenho da arquitetura para problemas reais com grandes volumes de dados. Outros trabalhos incluem avaliação da quantidade máxima de recursos físicos compatíveis com a quantidade de recursos disponíveis na placa Ultra96-V1 e testes de performance da solução CPU-FPGA para exploração de espaço de projeto na variação dos parâmetros do algoritmo para problemas com grande volume de dados.

6. AGRADECIMENTOS

Deixo aqui registrado o apoio da UFERSA e do CNPq através da bolsa de iniciação científica provinda do edital UFERSA-PROPPG 15/2023, que possibilitou a realização dessa pesquisa. Também agradeço aos demais alunos no grupo de pesquisa, em especial Gustavo Mendes.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[1] JAIN, A. Data clustering: 50 years beyond K-means. Pattern Recognition Letters, v.31, p.651-666, 2010.
[2] HAN, J.; KAMBER, M.; PEI, J. Data mining: concepts and techniques. Burlington, Ma: Elsevier, 2022.
[3] KUON, I.; TESSIER, R.; ROSE, J. FPGA Architecture: Survey and Challenges. Foundations and Trends® in Electronic Design Automation, v. 2, n. 2, p. 135-253, 2008.
[4] LAMATTICE SEMICONDUCTOR, What is an FPGA?. Disponível em <https://www.latticesemi.com/en/What-is-an-FPGA>. Acesso em: 23/04/2024.
[5] LLOYD, S. Least squares quantization in PCM, IEEE transactions on information theory 28.2: 129-137, 1982.
[6] MACIEL, L. A.; SOUZA, M. A.; FREITAS, H. C. de. Reconfigurable FPGA-Based K-Means/K-Modes Architecture for Network Intrusion Detection. IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, no. 8, pp. 1459-1463, 2020.
[7] SILVA, L. B. da *et al.* HPyC-FPGA - Integração de Aceleradores em FPGA de Alto Desempenho com Python para Jupyter Notebooks. Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho, Florianópolis, pp. 193-204, 2022.
[8] BRAGANÇA, L. *et al.* An Open Source Custom K-means Generator for AWS Cloud FPGA Accelerators. Anais do XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais, Evento Online, 2021, pp. 173-180.
[9] DIAS, L. A.; FERREIRA, J. C.; FERNANDES, M. A. C. Parallel Implementation of K-Means Algorithms on FPGA. IEEE Access, vol. 8, pp. 41071-41084, 2020.
[10] NETO, A. F. A.. Acelerador do cálculo distância euclidiana em Hardware. Trabalho de Conclusão de Curso, UFERSA, Mossoró: 2020.
[11] GUIMARÃES, G. M. Projeto de um Hardware Acelerador do Algoritmo de Distância Euclidiana. 2023. Trabalho de Conclusão de Curso. (Graduação em Ciência da Computação) - UFERSA, Mossoró: 2023.
[12] CARVALHO, J. B. P. 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.
[13] EDA PLAYGROUND: Disponível em: <https://www.edaplayground.com/>. Acesso em: 12/04/2024