IMPLEMENTAÇÃO DE FRACTAL EM FPGA USANDO O PROCESSADOR *SOFT-CORE* SAPHO

Luciano Manhães de Andrade Filho[1] - luciano.andrade@ufjf.br
Chrysthofer Arthur Amaro Afonso[2] - chrysthofer.afonso@estudante.ufjf.br
Eder Barboza Kapisch[3] - eder.kapisch@ufjf.br
Ana Luisa Basilio de Castro[4] - anabasilio.castro@estudante.ufjf.br
Sarita de Miranda Rimes[5] - smrimes@iprj.uerj.br
Gustavo Barbosa Libotte[6] - gustavolibotte@iprj.uerj.br
[1,2,3,4]Universidade Federal de Juiz de Fora – MG
[5,6]Instituto Politécnico do Rio de Janeiro (IPRJ/UERJ) – Nova Friburgo – RJ

Resumo.

O uso de fractais no processamento e na análise de dados tem ganhado destaque em diversas áreas científicas e tecnológicas, impulsionado pela necessidade de modelagem complexa e extração de padrões em sistemas dinâmicos. Projetos recentes na área de física, biomedicina e criptografia demonstram a relevância do cálculo eficiente de fractais como os de Weierstrass–Mandelbrot, Julia, Cantor dentre outros, cuja complexidade computacional exige soluções de alto desempenho. Nesse contexto, as Field-Programmable Gate Arrays (FPGAs) surgem como plataformas ideais para implementações embarcadas e paralelizadas, oferecendo flexibilidade, baixa latência e eficiência energética. Dentro desse cenário, o Scalable Architecture Processor for Hardware Optimization (SAPHO) se destaca como um processador soft-core projetado com lógica dedicada ao tratamento de números complexos, essenciais no cálculo de fractais. Neste trabalho, é apresentada a implementação do fractal do Conjunto de Mandelbrot utilizando o processador SAPHO em FPGA, explorando sua arquitetura reconfigurável para viabilizar o processamento rápido e eficiente de imagens fractais em tempo real. Além disso, propõe-se otimizações no código Assembly gerado para o SAPHO, que resultam em um ganho de desempenho de aproximadamente 25% em relação à implementação padrão, demonstrando o potencial da arquitetura para aplicações científicas que demandam alto desempenho com baixo consumo de recursos.

Keywords: *Fractais, FPGA, Processamento Embarcado, Processador Soft-Core, Computação de Alto Desempenho*

1. INTRODUÇÃO

Fractais são estruturas geométricas complexas cuja forma se repete em diferentes escalas, exibindo auto-semelhança e dimensão não inteira, propriedades que os tornam poderosas ferramentas para modelar fenômenos naturais e sistemas dinâmicos em diversas áreas do conhecimento. Introduzidos formalmente por Benoît B. Mandelbrot (1982), os fractais têm sido aplicados com sucesso em campos tão variados quanto a física, biologia, medicina, ciência da computação e arte, como demonstram Shu and Chan (2025) e Husain et al. (2022),

devido à sua capacidade de representar padrões irregulares e não lineares que desafiam as geometrias clássicas. Em métodos computacionais, os fractais têm se destacado em tarefas específicas: compressão de imagens (Li and Tak U, 2025), análise de sinais biomédicos (Yoder et al., 2024), simulação de ambientes naturais (Majhor and Bos, 2025) e criptografia baseada em fractais (Inam et al., 2025). Sua implementação eficiente exige, no entanto, recursos computacionais otimizados, especialmente quando aplicados em tempo real ou em sistemas embarcados, impulsionando o desenvolvimento de arquiteturas dedicadas para o processamento acelerado dessas estruturas matemáticas.

Um *Filed-Programmable Gate Array* (FPGA) é um circuito integrado reprogramável, geralmente baseado em memória volátil, com configuração descrita através de *Hardware Description Language* (HDL). Suas características fornecem uma solução versátil e poderosa para o desenvolvimento de circuitos dedicados ao processamento de tarefas intensivas. O paralelismo oferece a possibilidade de desenvolver circuitos específicos, em pipeline, operando múltiplas tarefas simultaneamente, o que aliado a operação em frequências elevadas, contribui para uma execução veloz e eficiente de sistemas digitais complexos. Assim, ao desenvolver o próprio hardware em HDL na FPGA, é obtido ótimo desempenho de processamento com baixo consumo de energia, sendo possível acelerar em até cem vezes o processamento da técnicas comparado as implementações convencionais em software (Arucu and Iliev, 2025).

A integração entre fractais e FPGAs surge como uma solução natural frente às altas demandas computacionais associadas ao processamento dessas estruturas matemáticas. Trabalhos nessa área têm sido cada vez mais postos em evidência, como nos projetos de geração do conjunto de Mandelbrot em FPGA (Armagost and Yang, 2008), compressão fractal de imagens de alta velocidade (Saad and Abdullah, 2018), análise e implementação de formas semi-fractais em hardware reconfigurável (AboAlNaga et al., 2021), e uso de junções Josephson fractais em geradores caóticos de números aleatórios (Radwan et al., 2020), que exploram diferentes formas de aceleração e implementação de algoritmos fractais em FPGA. Além disso, a eficiência energética alcançada por circuitos dedicados em FPGA torna essa tecnologia particularmente atrativa para sistemas embarcados e aplicações que requerem operação contínua ou em larga escala (Mittal, 2017). Dessa forma, o desenvolvimento de arquiteturas de processamento de fractais em FPGA não apenas amplia o alcance prático de suas aplicações, mas também estabelece uma ponte estratégica entre teoria matemática e soluções de engenharia de alto impacto tecnológico.

Uma forma eficiente de desenvolvimento de circuitos complexos em FPGA é o uso de processadores embarcados Soft-Core. Neste contexto, o processador *Scalable Architecture Processor for Hardware Optimization* (SAPHO), desenvolvido pelo Núcleo de Instrumentação e Processamento de Sinais (NIPS) da Universidade Federal de Juiz de Fora (UFJF)[1], apresenta uma arquitetura baseada no modelo Harvard, oferecendo um conjunto reduzido de instruções parametrizável de acordo com o código a ser executado (Stallings, 2009). Sua flexibilidade se evidencia no ajuste automático do tamanho das Memórias de Dados (DM) e Memórias de Programa (PM), bem como na configuração da Unidade Lógico-Aritmética (ULA), com largura de palavra adaptável às necessidades do projeto. Desenvolvido integralmente em Verilog HDL, o SAPHO pode ser sintetizado em qualquer dispositivo FPGA, possibilitando portabilidade e otimização do hardware. Entre suas vantagens destacam-se a alocação escalável de recursos, com ALU combinacional capaz de executar uma instrução por ciclo de clock sem quebra de pipeline. Sua linguagem de programação, otimizada para cálculo com números complexos, o torna particularmente adequado para a implementação de fractais.

Dessa forma, o objetivo deste trabalho é utilizar e adaptar o processador SAPHO para o desenvolvimento de técnicas de processamento de fractais em FPGA, estabelecendo um framework que possa servir como base para futuras pesquisas e aplicações nessa área.

[1] https://www.nipscern.com/

A proposta busca não apenas validar a viabilidade do uso do SAPHO em algoritmos matematicamente intensivos, mas também oferecer uma plataforma flexível e escalável para o estudo e a implementação de diferentes tipos de fractais em hardware reconfigurável. Como prova de conceito, será apresentada a implementação do fractal de Mandelbrot, ilustrando o potencial da abordagem e evidenciando os ganhos de desempenho e eficiência proporcionados pela integração entre arquiteturas dedicadas e o processamento de fractais em tempo real.

O trabalho está estruturado da seguinte forma: Na Seção 2 são apresentados os fractais e o conjunto de MandelBrot. Na Seção 3, é descrito o processador soft-core SAPHO utilizado na implementação do trabalho, a qual é descrita na Seção 4. Já na Seção 5, são apresentados resultados. Por fim, na Seção 6, é feita uma conclusão.

2. FRACTAIS E O CONJUNTO DE MANDELBROT

Somos expostos cotidianamente às noções intuitivas de dimensões espaciais tal como tratadas na geometria euclidiana (Euclid and Heath, 1956). Euclides e, posteriormente, Riemann ampliaram essas concepções ao estabelecer que espaços de dimensão n podem ser descritos em $\mathbb{R}^n$, formalizando variedades e geometrias intrínsecas (Riemann, 2004). Na tradição euclidiana, um ponto não possui extensão; uma reta possui comprimento; um plano é caracterizado por sua área; e o espaço tridimensional é descrito por volume (Euclid and Heath, 1956). Uma mudança de paradigma veio quando Felix Hausdorff introduziu a noção técnica de medida e dimensão fracionária (Hausdorff, 1918). Décadas mais tarde, Mandelbrot mostrou que estruturas naturais exibem propriedades de auto-semelhança, complexidade infinita e dimensões não inteiras (Mandelbrot, 1967).

O termo *fractal* foi introduzido por Benoît B. Mandelbrot a partir do latim *fractus*, que significa ”fragmentado”ou ”irregular”. A escolha do termo reflete a ideia de formas geométricas que não se ajustam a descrições clássicas por dimensões inteiras e apresentam estruturas complexas em múltiplas escalas (Mandelbrot, 1982). Um conceito fundamental é o de **auto-semelhança**: um objeto é dito auto-semelhante quando suas partes, em diferentes escalas de ampliação, apresentam a mesma estrutura global. Além disso, fractais caracterizam-se por exibir **complexidade infinita**, isto é, detalhes que persistem independentemente do grau de ampliação, e por possuírem uma **dimensão fractal**, frequentemente não inteira, que quantifica precisamente tal irregularidade (Falconer, 2014). Esses três elementos — auto-semelhança, complexidade infinita e dimensão fracionária — compõem a essência da definição de um fractal.

Um dos fractais de maior interesse é o conjunto de Mandelbrot. Ele é definido pelo estudo da iteração da função complexa

$$z_{n+1} = z_n^2 + c \tag{1}$$

onde $z_0 = 0$ e $c \in \mathbb{C}$ é um parâmetro complexo. O conjunto de Mandelbrot é composto pelos valores de c para os quais a sequência (z_n) permanece limitada. Nesse contexto:

- z_n representa a iteração no plano complexo,
- c é o parâmetro que varia e determina a forma global do conjunto,
- a condição de *limitação* distingue se c pertence ou não ao conjunto.

A fronteira do conjunto de Mandelbrot é infinitamente detalhada e exibe auto-semelhança em várias escalas, como mostra a Figura 1.

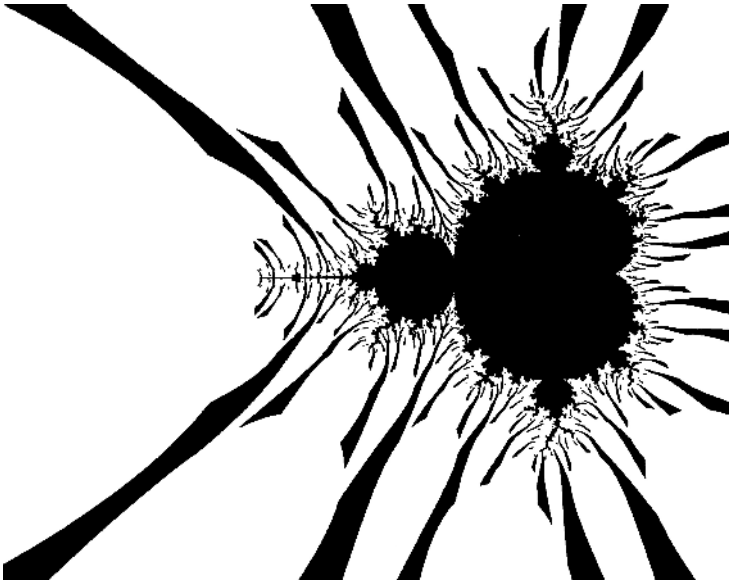


Figura 1: Imagem do fractal do conjunto de Mandelbrot evidenciando a complexidade infinita

3. PROCESSADOR SOFT-CORE - SAPHO

O SAPHO é um ambiente integrado de desenvolvimento voltado para a criação, parametrização, programação, compilação, sintetização, depuração e análise de processadores *soft-core* em FPGA para múltiplas aplicações. O SAPHO conta com uma *Integrated Development Environment* (IDE) repleta de recursos para desenvolvimento dos processadores, denominada AURORA.

A estrutura dos processadores desenvolvidos no SAPHO baseia-se em uma arquitetura escalável e modular, onde o usuário pode definir as características do processador diretamente pela IDE AURORA durante a criação. A AURORA permite a programação utilizando uma linguagem derivada da linguagem C, batizada de `C±` (também CMM), ou diretamente em Assembly, facilitando o desenvolvimento e otimização de códigos para aplicações embarcadas em FPGA. O código escrito nessas linguagens é compilado e convertido automaticamente em código HDL (Verilog), que descreve o processador em nível de hardware de forma transparente ao projetista. Outro recurso que diferencia o SAPHO é o suporte a tipos de dados específicos: além dos tradicionais `int` e `float`, comuns em C, a arquitetura oferece o tipo `comp`, projetado para operações nativas com números complexos, recurso essencial em aplicações como o cálculo de fractais, análise espectral e processamento de sinais multidimensionais.

Uma vez que os blocos de hardware em Verilog foram produzidos como resultado da compilação dos códigos de programa pela AURORA, os mesmos podem ser utilizados por sistemas de processamento digitais complexos e interconectados dentro da própria AURORA, ou ser importados para outros sintetizadores, como o Quartus II® da Intel®, visando a compatibilidade com outros ambientes sintetizadores.

Uma das principais vantagens do SAPHO é a capacidade de sintetizar apenas os blocos de hardware necessários para a execução do programa. Os comandos utilizados no código de programa que será carregado ao processador determinam quais recursos serão instanciados na FPGA, como as operações da ULA, registradores de uso específico, interfaces de memória, inclusive operações em ponto flutuante. Assim, evita-se o desperdício de lógica programável e otimiza-se o uso de recursos do chip. Dessa forma, cada processador gerado é customizado de acordo com as necessidades da aplicação, garantindo eficiência energética e otimização de consumo de lógica na implementação em FPGA. Diferentemente de arquiteturas convencionais de processadores *soft-core*, o SAPHO incorpora um mecanismo de alocação automática de recursos de hardware durante o processo de compilação, permitindo a geração de circuitos digitalmente otimizados de acordo com as operações especificadas no código-fonte do usuário.

Dessa forma, processadores SAPHO destacam-se pelo seu alto desempenho no processamento de grandes volumes de dados ao mesmo tempo que requerem baixo consumo de recursos computacionais (Viccini, 2023).

Outro aspecto relevante da arquitetura do SAPHO é sua capacidade de executar instruções em um único ciclo de clock quando configurado com um pipeline de três estágios: **F** – Fetch (busca da instrução), **D** – Decode (decodificação) e **E** – Execute (execução). Essa organização mantém a cadeia de execução contínua (**F** $\rightarrow$ **D** $\rightarrow$ **E**), mesmo em programas que contenham instruções condicionais ou chamadas de função. Além disso, quando o circuito digital incorpora operações mais complexas, como divisões em hardware, multiplicações de alta precisão ou funções matemáticas especiais, o usuário pode aumentar o número de estágios do pipeline. Essa flexibilidade não altera a compatibilidade do programa, mas possibilita maior desempenho e melhor aproveitamento dos recursos da FPGA, adaptando a profundidade do pipeline ao grau de complexidade do processador.

A interface de programação é o principal ponto de interação entre o usuário e os processadores desenvolvidos no SAPHO. A Figura 2 mostra uma tela da versão da IDE AURORA utilizada neste trabalho. Na parte superior, há ferramentas para criação, parametrização e compilação dos processadores; na parte esquerda, uma árvore de projeto com instâncias criadas; e na tela principal, um código em `C±` utilizado para o desenvolvimento do processador FFPGA.

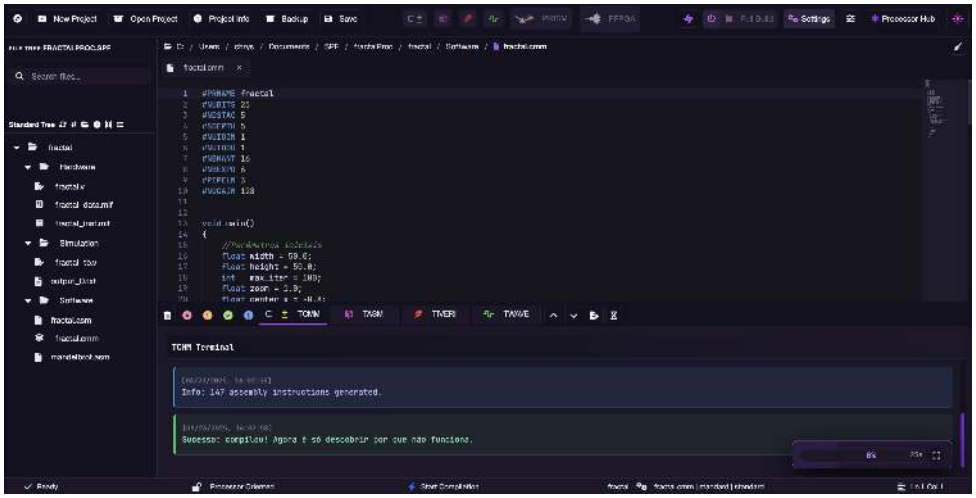


Figura 2: Imagem da versão da IDE do SAPHO, AURORA, utilizada neste trabalho.

A IDE do SAPHO oferece uma interface intuitiva e funcional, permitindo a criação de projetos com nome e local, além de abrir projetos `.spf` existentes. Ela inclui um compilador `C±` que traduz o código CMM para ASM, e um compilador ASM que converte o código para Verilog, a linguagem de descrição de hardware. A IDE também suporta a simulação com o Icarus Verilog (`iverilog`) e a visualização dos sinais da simulação com o GTKWave. O PRISM, um visualizador RTL dentro da interface AURORA, exibe a arquitetura do processador. Todas essas etapas podem ser automatizadas, incluindo a configuração de parâmetros como frequência de clock e escolha do testbench. A criação do processador permite definir características como número de bits, mantissa, expoente e tamanhos das pilhas de memória e dados. A estrutura do projeto é visualizada em árvore de forma modular, e o editor Monaco oferece uma experiência similar ao Visual Studio Code. O terminal separa as ferramentas de compilação e simulação, e a barra de progresso com *timer* indica o tempo de cada processo.

É importante observar que esta IDE está em constante atualização, recebendo novos recursos e aprimoramentos dentro do projeto de desenvolvimento do SAPHO realizado no NIPS. A versão apresentada neste trabalho pode diferir ligeiramente daquela encontrada no repositório do projeto.

4. IMPLEMENTAÇÃO

A implementação do conjunto de Mandelbrot na linguagem `C±` inicia-se pela definição de parâmetros relacionados diretamente à geração da imagem do fractal, como mostrado na Figura 3. A largura (`width`) e altura (`height`) definem a resolução em pixels, influenciando o nível de detalhe visível. Um aumento nesses valores amplia a precisão visual, mas também eleva o custo computacional, já que mais pontos precisam ser iterados. O parâmetro `max_iter` representa o limite de iterações por pixel: valores baixos produzem imagens menos detalhadas, enquanto valores mais altos permitem explorar nuances mais finas da fronteira do conjunto, resultando em maior riqueza visual.

Os limites da janela (`x_min`, `x_max`, `y_min`, `y_max`) determinam a região do plano complexo a ser mapeada. Alterar esses valores equivale a aplicar zoom ou translação na imagem final, permitindo explorar regiões específicas do fractal. A correspondência entre coordenadas de pixel e coordenadas complexas é realizada por meio das escalas `x_scale` e `y_scale`, que ajustam a proporção de cada passo em função da resolução escolhida. Assim, cada pixel da imagem está associado a um ponto $c \in \mathbb{C}$.

```
12  void main()                 // ponto de entrada
13  {
14      // Parâmetros iniciais vindos do XML
15      float width = 800.0;      // largura da imagem (px)
16      float height = 600.0;     // altura da imagem (px)
17      int   max_iter = 1023;    // iterações máximas por pixel
18
19      // Limites da janela do fractal e, portanto, da região de iteração
20      float x_min = -3.047291359679012049389;  // limite esquerdo (eixo real)
21      float x_max =  1.560661232913579950611;  // limite direito (eixo real)
22      float y_min = -1.596716255538864018556;  // limite inferior (eixo imag.)
23      float y_max =  1.859248198913580067844;  // limite superior (eixo imag.)
24
25      // Escalas para transformar pixels em coordenadas complexas
26      float x_scale = (x_max - x_min) / (width - 1);   // passo em x por pixel
27      float y_scale = (y_max - y_min) / (height - 1);  // passo em y por pixel
28
29      int px = 0;                 // coluna atual (x)
30      int py = 0;                 // linha atual (y)
31
```

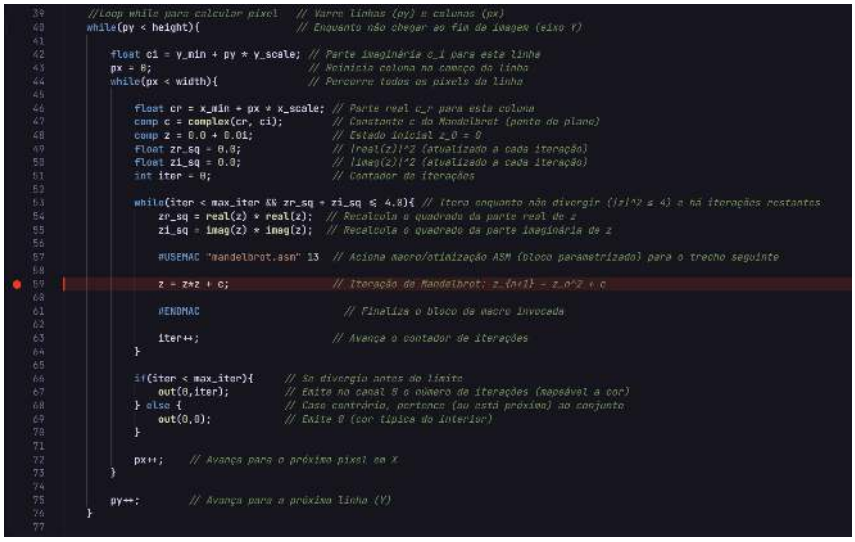
Figura 3: Parâmetros iniciais em código `cmm`, definindo resolução, limites da janela e número máximo de iterações.

O núcleo da execução é o laço duplo `while`, responsável por varrer todos os pixels da imagem, linha por linha. O primeiro `while` percorre as linhas verticais (`py`), e o segundo percorre as colunas (`px`). Para cada pixel, calcula-se o ponto complexo $c = cr + i\,ci$, a partir dos limites da janela e das escalas definidas. Em seguida, inicializa-se $z_0 = 0$, e aplica-se iterativamente a relação fundamental 1.

A condição de parada do laço interno de iteração combina dois critérios: o número de iterações ainda não ter ultrapassado `max_iter`, e o módulo $|z|^2$ permanecer menor ou igual a 4. O primeiro critério limita o custo computacional, enquanto o segundo garante a distinção entre pontos que divergem e aqueles que permanecem limitados. Quando $|z|^2 > 4$, o ponto c é considerado fora do conjunto, e a contagem de iterações até a fuga é registrada, como explicitado na Figura 4.

A Figura 5 mostra um aspecto distintivo da linguagem `.cmm` que é o uso da diretiva #USEMAC, a qual permite substituir blocos de código de alto nível por versões otimizadas em `ASM`. No caso analisado, o trecho de iteração do fractal foi encapsulado em #USEMAC e substituído pelo código `mandelbrot.asm`. Esse recurso garante que a lógica matemática permaneça intacta, mas a execução ocorra de maneira mais eficiente, aproveitando instruções específicas em baixo nível.

Por fim, após a iteração de cada pixel, o programa emite o resultado pela instrução `out(0, iter)`. Quando o ponto diverge antes do limite, o valor de `iter` é escrito; caso contrário, é registrado o valor zero, indicando pertencimento ao conjunto. Essa saída é gravada sequencialmente em um arquivo `output_0.txt`, no qual cada linha corresponde à cor de um pixel da imagem, organizada da esquerda para a direita e de cima para baixo. Posteriormente, o programa `fancyFractal` interpreta esse arquivo, aplicando mapeamentos



```
//Loop while para calcular pixel   // Varre linhas (py) e colunas (px)
while(py < height){               // Enquanto não chegar ao fim da imagem (eixo Y)

    float ci = y_min + py * y_scale; // Parte imaginária c_i para esta linha
    px = 0;                          // Reinicia coluna no começo da linha
    while(px < width){               // Percorre todos os pixels da linha

        float cr = x_min + px * x_scale; // Parte real c_r para esta coluna
        comp c = complex(cr, ci);        // Constante c do Mandelbrot (ponto do plano)
        comp z = 0.0 + 0.0i;             // Estado inicial z_0 = 0
        float zr_sq = 0.0;               // real(z)^2 (atualizado a cada iteração)
        float zi_sq = 0.0;               // imag(z)^2 (atualizado a cada iteração)
        int iter = 0;                    // Contador de iterações

        while(iter < max_iter && zr_sq + zi_sq <= 4.0){ // Itera enquanto não divergir (|z|^2 ≤ 4) e há iterações restantes
            zr_sq = real(z) * real(z);  // Recalcula o quadrado da parte real de z
            zi_sq = imag(z) * imag(z);  // Recalcula o quadrado da parte imaginária de z

            #USEMAC "mandelbrot.asm" 13  // Aciona macro/otimização ASM (bloco parametrizado) para o trecho seguinte

            z = z*z + c;                 // Iteração de Mandelbrot: z_{n+1} = z_n^2 + c

            #ENDMAC                      // Finaliza o bloco de macro invocada

            iter++;                      // Avança o contador de iterações
        }

        if(iter < max_iter){     // Se divergiu antes do limite
            out(0,iter);         // Exite no canal 0 o número de iterações (mapeável a cor)
        } else {                 // Caso contrário, pertence (ou está próximo) ao conjunto
            out(0,0);            // Exite 0 (cor típica do interior)
        }

        px++;    // Avança para o próximo pixel em X
    }

    py++;        // Avança para a próxima linha (Y)
}
```

Figura 4: Laço duplo de varredura e iteração $z = z * z + c$, núcleo do cálculo do conjunto de Mandelbrot.

```
LOD main_z
P_LOD main_z
F_MLT main_z
P_LOD main_z_i
F_MLT main_z_i
F_NEG
SF_ADD
F_ADD main_c
SET_P main_z
F_MLT main_z_i
F_MLT 2.0
F_ADD main_c_i
SET main_z_i
```

Figura 5: Trecho otimizado em `ASM` (`mandelbrot.asm`) incluído por meio da diretiva #USEMAC.

de cor, suavizações e estilos visuais personalizados para produzir a imagem final do fractal, pronta para visualização e análise.

Houve uma mudança significativa no framework da IDE AURORA para dar suporte ao projeto do fractal. As principais alterações incluem a criação do `Fractal Viewer` e a implementação do botão FFPGA, responsável por gerenciar todo o pipeline de compilação, desde a criação dos códigos intermediários até a visualização dos resultados no `Fractal Viewer`. A Figura 6 apresenta a interface do `Fractal Viewer`, onde os resultados do fractal são exibidos após a compilação e execução do código, podendo ser realizados ajustes finos de imagem.

5. RESULTADOS

Utilizando uma configuração de FPGA convencional, com frequência de operação de 600 MHz, realizamos a simulação do código no ambiente GTKWave. O tempo total de execução do código original foi de aproximadamente 57 ms (ou 56726929.315 ns).

Após aplicar a diretiva #USEMAC e otimizar o código com o arquivo `mandelbrot.asm`, mantendo a mesma frequência de 600 MHz, o tempo de execução foi reduzido para cerca de 51 ms (51216331.103 ns). Essa melhoria representa uma redução de aproximadamente 10.6% no tempo total de simulação, destacando os benefícios da otimização em baixo nível.

Além da diminuição do tempo de execução, vale ressaltar que a FPGA oferece amplas oportunidades para paralelização. Sua arquitetura possibilita a replicação massiva de núcleos de cálculo, permitindo que centenas de unidades sejam instanciadas em paralelo. Cada núcleo

Figura 6: Fractal Viewer com fractal aberto para edição.

poderia ser responsável por calcular um segmento específico da imagem fractal, o que reduziria significativamente o tempo total de processamento.

A implementação no SAPHO, com otimização no código assembly, obteve um tempo de execução de 51 ms, superando os 53 ms da tradução fiel do código `C±` para C, executado em um computador convencional. Este resultado, alcançado com uma única otimização, gera otimismo sobre o potencial da abordagem FPGA para futuras melhorias no tempo de execução.

Para efeito de análise, apresentamos a seguir três figuras: (7) a simulação completa do `gtkwave`, (8) um recorte ampliado mostrando o laço principal e a quantidade de ciclos necessários na versão original, e (9) a mesma região após a otimização, onde observa-se uma redução de aproximadamente 25% no número de ciclos do laço principal.

Figura 7: Simulação completa no GTKWave do cálculo do fractal.

6. CONCLUSÃO

Este trabalho apresentou a implementação do conjunto de Mandelbrot em FPGA utilizando o processador SAPHO, explorando a flexibilidade da arquitetura reconfigurável e sua linguagem dedicada ao cálculo de números complexos. Os resultados obtidos evidenciam que, em uma configuração convencional de 600 MHz, é possível alcançar desempenho elevado com baixo consumo de recursos, sendo que a otimização em Assembly reduziu em aproximadamente 10,6% o tempo total de execução e em torno de 25% o número de ciclos no laço principal.

Com isso, consolidamos um *framework* funcional para o processamento de fractais em FPGA, que pode ser diretamente aplicado em diferentes áreas científicas e tecnológicas, como

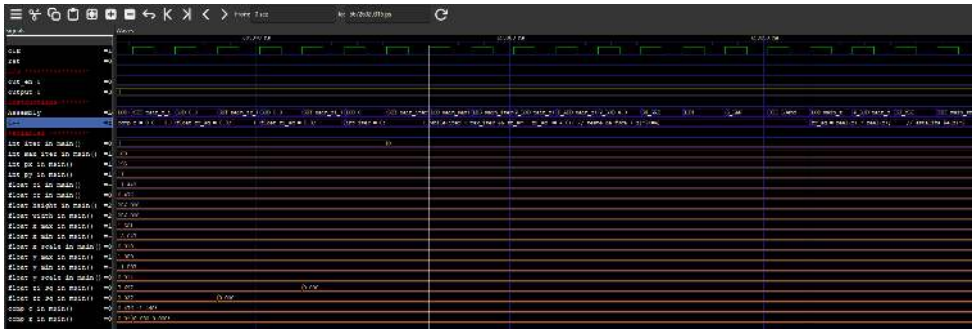


Figura 8: Detalhe do laço principal no código original, com número total de ciclos indicado.

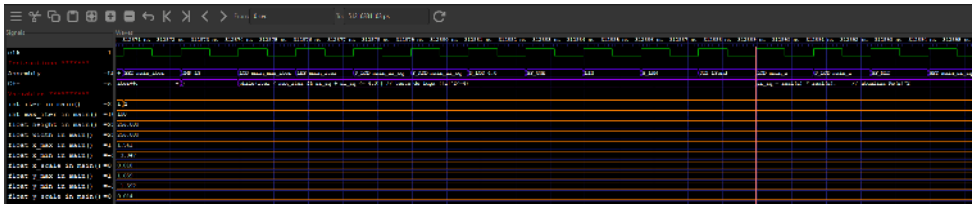


Figura 9: Laço principal otimizado no código `mandelbrot.asm`, reduzindo em cerca de 25% o número de ciclos.

física, biomedicina, criptografia e análise de sinais. Além disso, a capacidade intrínseca de paralelização das FPGAs abre caminho para expansões futuras, permitindo a criação de múltiplos núcleos de cálculo dedicados e possibilitando ganhos ainda mais expressivos de desempenho.

Acknowledgements

Os autores agradecem ao Núcleo de Instrumentação e Processamento de Sinais (NIPS) da Universidade Federal de Juiz de Fora pelo suporte técnico e acadêmico que possibilitou o desenvolvimento deste trabalho. Agradecemos também ao CNPq pelo apoio financeiro, que tem sido fundamental para a continuidade das pesquisas.

Referências

AboAlNaga, B. M. et al. (2021). Analysis and fpga of semi-fractal shapes based on complex gaussian map. *International Journal of Electrical and Computer Engineering (IJECE)*, 11(5):4324–4334.

Armagost, J. and Yang, E. (2008). A mandelbrot set generator implemented on an altera de1 board. Technical report, University of California, Berkeley. Course Project Report.

Arucu, M. and Iliev, T. (2025). Performance evaluation of FPGA, GPU, and CPU in FIR filter implementation for semiconductor-based systems. *Journal of Low Power Electronics and Applications*, 15(3):40.

Euclid and Heath, T. L. t. (1956). *The Elements*. Dover Publications, New York. Translation by T. L. Heath; original work: Euclid (c. 300 BCE).

Falconer, K. J. (2014). *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, Chichester, UK, 3 edition.

Hausdorff, F. (1918). Dimension und äußeres maß. *Mathematische Annalen*, 79:157–179.

Husain, A., Nanda, M. N., Chowdary, M. S., and Sajid, M. (2022). Fractals: an eclectic survey, part-i. *Fractal and Fractional*, 6(2):89.

Inam, S., Kanwal, S., Batool, M., Al-Otaibi, S., and Jamjoom, M. M. (2025). A blockchain-integrated chaotic fractal encryption scheme for secure medical imaging in industrial iot settings. *Scientific Reports*, 15(1):7652.

Li, M. and Tak U, K. (2025). An enhanced fractal image compression algorithm based on adaptive non-uniform rectangular partition. *Electronics*, 14(13):2550.

Majhor, C. D. and Bos, J. P. (2025). Multifractal terrain generation for evaluating autonomous off-road ground vehicles. *Journal of Autonomous Vehicles and Systems*, 5(2):021003.

Mandelbrot, B. B. (1967). How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, 156(3775):636–638.

Mandelbrot, B. B. (1982). *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York, NY. Clássico fundacional dos fractais; introduz auto-semelhança, complexidade infinita e dimensão fracional.

Mittal, S. S. (2017). Energy-efficient FPGA-based embedded systems: A survey of trends, techniques, and applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–27.

Radwan, A. G. et al. (2020). Dynamical analysis, fpga implementation and its application to chaos-based random number generator of a fractal josephson junction. *IEEE Access*, 8:164750–164762.

Riemann, B. (2004). *Collected Papers / Gesammelte Mathematische Werke*. Springer, Berlin, Germany. Includes translation and commentary of the 1854 lecture "Über die Hypothesen, welche der Geometrie zu Grunde liegen".

Saad, A.-M. and Abdullah, M. Z. (2018). High-speed fractal image compression featuring deep data pipelining strategy. *International Journal of Engineering & Technology (UAE)*, 7(4.31):36–40.

Shu, Z. and Chan, P. W. (2025). Application of fractal analysis on wind speed time series: a review. *Advances in Wind Engineering*, page 100028.

Stallings, W. (2009). *Arquitetura e Organização de Computadores*. Pearson Universidades, 8 edition.

Viccini, L. (2023). Otimizações para processador soft-core embarcado em FPGA visando implementação de métodos de reconstrução online de energia em aceleradores de partículas. Dissertação (mestrado), Universidade Federal de Juiz de Fora.

Yoder, K. J., Brookshire, G., Glatt, R. M., Merrill, D. A., Gerrol, S., Quirk, C., and Lucero, C. (2024). Fractal dimension distributions of resting-state electroencephalography (eeg) improve detection of dementia and alzheimer's disease compared to traditional fractal analysis. *Clinical and Translational Neuroscience*, 8(3):27.

Abstract. *The use of fractals in data processing and analysis has gained prominence in various scientific and technological fields, driven by the need for complex modeling and pattern extraction in dynamic systems. Recent projects in physics, biomedicine, and cryptography have demonstrated the relevance of efficient fractal computation, such as Weierstrass–Mandelbrot, Julia, and Cantor sets, whose computational complexity requires high-performance solutions. In this context, Field-Programmable Gate Arrays (FPGAs) emerge as ideal platforms for embedded and parallelized implementations, offering flexibility, low latency, and energy efficiency. Within this scenario, the Scalable Architecture Processor for Hardware Optimization (SAPHO) stands out as a soft-core processor designed with dedicated logic for handling complex numbers, which are essential in fractal computation. In this work, we present the implementation of the Mandelbrot Set fractal using the SAPHO processor on FPGA, exploring its reconfigurable architecture to enable fast and efficient real-time fractal image processing. Furthermore, we propose optimizations in the Assembly code generated for SAPHO, which result in a performance improvement of approximately 25% compared to the standard implementation, demonstrating the potential of the architecture for scientific applications that demand high performance with low resource consumption.*

Keywords: *Fractals, FPGA, Embedded Processing, Soft-Core Processor, High-Performance Computing*