Universidade Federal do Espírito Santo
Departamento de Engenharia Elétrica
Programa de Pós-Graduação em Engenharia Elétrica

Lucas Castro de Rezende

COORDENAÇÃO DE CARREGAMENTO DE VEÍCULOS ELÉTRICOS ATRAVÉS DE SIMULAÇÃO EM TEMPO REAL E OTIMIZAÇÃO PELO MÉTODO DE ALGORITMOS GENÉTICOS

Vitória – ES

Agosto de 2022

Lucas Castro de Rezende

COORDENAÇÃO DE CARREGAMENTO DE VEÍCULOS ELÉTRICOS ATRAVÉS DE SIMULAÇÃO EM TEMPO REAL E OTIMIZAÇÃO PELO MÉTODO DE ALGORITMOS GENÉTICOS

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Universidade Federal do Espírito Santo como parte dos requisitos necessários para a obtenção do Grau de Mestre em Engenharia Elétrica.

Área de Concentração: Processamento de Energia e Sistemas Elétricos

Augusto César Rueda Medina, Dr.

Orientador

Vitória – ES

Agosto de 2022

Ficha catalográfica disponibilizada pelo Sistema Integrado de Bibliotecas - SIBI/UFES e elaborada pelo autor

---

Rezende, Lucas Castro de, 1994-
R467c Coordenação de carregamento de veículos elétricos através de simulação em tempo real e otimização pelo método de Algoritmos Genéticos / Lucas Castro de Rezende. - 2022.
93 f. : il.

Orientador: Augusto César Rueda Medina.
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Engenharia elétrica. 2. Veículos elétricos. 3. Algoritmos genéticos. 4. Arranjos de lógica programável em campo. 5. Programação paralela (Computação). I. Medina, Augusto César Rueda. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 621.3

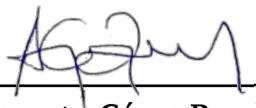
Lucas Castro de Rezende

COORDENAÇÃO DE CARREGAMENTO DE VEÍCULOS ELÉTRICOS ATRAVÉS DE SIMULAÇÃO EM TEMPO REAL E OTIMIZAÇÃO PELO MÉTODO DE ALGORITMOS GENÉTICOS

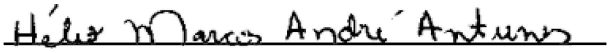
Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Universidade Federal do Espírito Santo como parte dos requisitos necessários para a obtenção do Grau de Mestre em Engenharia Elétrica.

Aprovada em 17 de Agosto de 2022.

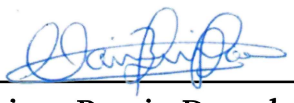
COMISSÃO EXAMINADORA



Prof. Dr. Augusto César Rueda Medina
Universidade Federal do Espírito Santo
Orientador



Prof. Dr. Hélio Marcos André Antunes
Universidade Federal do Espírito Santo
Examinador Interno



Prof. Dr. Clainer Bravin Donadel
Instituto Federal do Espírito Santo
Examinador Externo

Agradecimentos

Gostaria de agradecer aos meus familiares por me darem suporte em todo meu trajeto, desde sempre, demonstrando apoio e compreensão. Também gostaria de agradecer especialmente à minha companheira Leticia Cavassana que sempre esteve ao meu lado quando precisei, sendo compreensiva e me encorajando a continuar nessa árdua jornada de um mestre, nos momentos necessários, proporcionando-me conforto, solidariedade e sobretudo amor. Também gostaria de agradecer a Ufes e ao Departamento de Engenharia Elétrica pela oportunidade e em especial ao meu orientador Augusto César Rueda Medina, que me acolheu no mundo acadêmico e me guiou por essa caminhada repleta de desafios.

O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq).

"Nossas dúvidas são traidoras e nos fazem perder o bem que poderíamos conquistar, se não fosse o medo de tentar."

(William Shakespear)

RESUMO

Nos últimos anos, a preocupação em relação ao uso da energia de fontes limpas aumentou consideravelmente, resultando em diversas novas tecnologias e modelos de sistemas a fim de amenizar o efeito estufa relacionado à queima de combustíveis fósseis. Os veículos elétricos (VEs) e os geradores distribuídos (GDs) a partir de fontes renováveis são apontados como soluções para o problema; entretanto, a sua implementação no sistema pode causar adversidades no sistema de potência, caso feito de forma descoordenada. Neste presente estudo, teve-se o objetivo de elaborar uma solução a partir do uso do Método de Algoritmos Genéticos (AG), com auxílio do *software A Mathematical Programming Language* (AMPL) para obter um algoritmo que coordene, de forma otimizada, tanto o carregamento dos VEs, quanto a potência fornecida de cada GD em cada período de tempo, de forma a respeitar os limites operativos e de segurança do sistema de distribuição de energia elétrica, dos VEs e dos GDs. O método mostrou resultados satisfatórios em sua aplicação e reduziu o custo do sistema em aproximadamente 12,19% em relação ao sistema sem GDs e VEs. Na segunda etapa da pesquisa, foram apresentados métodos de paralelização de *hardware*, que consiste em aplicar tarefas específicas para um *hardware* dedicado, aumentando a velocidade de computação. A partir das demonstrações, foi aplicado em um AG e observado que, de fato, há ganhos consideráveis na velocidade de processamento, sendo um alternativa viável para implementar em conjunto com os AG.

Palavras-chave: Algoritmos Genéticos; Veículos Elétricos; *Harware-in-the-loop*; *High-Level Synthesis*; Métodos de paralelização de *hardware*.

ABSTRACT

In recent years, the concern regarding the use of energy from clean sources has increased considerably, resulting in several new technologies and system models in order to alleviate the greenhouse effect related to the burning of fossil fuels. Electric vehicles and distributed generators from renewable sources are pointed out as solutions to the problem; however, its implementation in the system can cause adversity in the power system, if done in an uncoordinated way. In this study, the objective was to develop a solution using the Genetic Algorithms Method, with the aid of the A Mathematical Programming Language (AMPL) software to obtain an algorithm that optimally coordinates both the loading of the electric vehicles, as the power supplied by each distributed generator in each period of time, in order to respect the operational and safety limits of the electric energy distribution system, of the electric vehicles and of the distributed generators. The method showed satisfactory results in its application and reduced the system cost by approximately 12.19% compared to the system without distributed generators and electric vehicles. In the second stage of the research, hardware parallelization methods were presented, which consists of applying specific tasks to a dedicated hardware, increasing the computation speed. From the demonstrations, it was applied in a GA and observed that, in fact, there are considerable gains in processing speed, being a viable alternative to implement together with the GA.

Keywords: Genetic Algorithm; Electric Vehicles, Hardware-in-the-loop; High-Level Synthesis; Hardware parallelization methods.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

Lista de abreviaturas e siglas

VE	Veículos Elétricos
GD	Geração Distribuída
AG	Algoritmos Genéticos
AMPL	*A Mathematical Programming Language*
HIL	*Hardware-in-the-loop*
HLS	*High-Level Synthesis*
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
FPO	Fluxo de potência ótimo
MPI	Método dos Pontos Interiores
HEV	Veículo Elétrico Híbrido
PHEV	Veículo Elétrico Híbrido *Plug-in*
PEV	Veículo Elétrico *Plug-in*
SoC	Estado de carga
FPGA	*Field Programmable Gate Array*
OTP	*One-time Programmable*
SRAM	*Static Random Acess Memory*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
RTL	*Register Transfer-Level*
CPU	*Central Processor Unit*
GPU	*Unidade de Processamento Gráfico*
CU	*Computing Unit*
FIFO	*First-In-First-Out*
SO	Sistema Operacional

Lista de símbolos

Y_{ij}	Negativo da admitância entre nós ij
Y_{ii}	Auto admitância do nó i
I_i	Corrente de injeção no nó i
V_i	Tensão no nó i
t	Instante de tempo (h)
T	Total de horas considerado (h)
Δt	Variação do instante de tempo (h)
$C_{ss}(t)$	Custo da potência ativa da subestação (R$/kW)
$P_{ss}(t)$	Potência ativa fornecida da subestação (kW)
$Q_{ss}(t)$	Potência reativa fornecida da subestação (kvar)
dg	Número da GD
DG	Número máximo de GDs
$C_{dg}(dg,t)$	Custo da potência ativa da GD (R$/kW)
$Pdg(dg,t)$	Potência ativa fornecida pela GD (kW)
$Pdg_{mx}(dg,t)$	Potência ativa máxima fornecida pela GD (kW)
$Qdg(dg,t)$	Potência reativa fornecida pela GD (kvar)
$F_{Pen_{ss}}$	Fator penalidade para subestação
OL_{ss}	Valor do fator de potência que ultrapassa o permitido
F_{Pen_V}	Fator penalidade para a tensão dos nós
N_{busses}	Número total de nós
$N_{branches}$	Número total de ramos do sistema
OL_V	Valor da tensão que ultrapassa o permitido (kV)

F_{Pen_I}	Fator penalidade para corrente do ramo
OL_I	Valor da corrente que ultrapassa o permitido (A)
$F_{Pen_{VE}}$	Fator de penalidade da carga do VE
VE	Número do veículo elétrico
N_{VE}	Número total de VEs
OL_{NC}	Potência fora dos limites de carregamento (kW)
$P(j,i,t)$	Potência ativa fluindo no ramo (kW)
$Q(j,i,t)$	Potência reativa fluindo no ramo (kvar)
$R(i,j)$	Resistência no ramo (kΩ)
$X(i,j)$	Reatância no ramo (kΩ)
$Z(i,j)$	Impedância no ramo (kΩ)
$I(i,j,t)$	Corrente fluindo no ramo(A)
$I_{mx}(i,j)$	Corrente máxima no ramo (A)
$V(j,t)$	Tensão no nó j (kV)
$V_{mx}(j,t)$	Tensão máxima no nó (kV)
$V_{min}(j,t)$	Tensão mínima no nó (kV)
$P_d(i)$	Potência ativa demandada no nó (kW)
$Q_d(i)$	Potência reativa demandada no nó (kvar)
$fd(t)$	Fator de demanda no tempo
$c(VE,j,t)$	Estado de conexão do VE no nó e no tempo
fp_{min}	Fator de potência mínimo
fp_{mx}	Fator de potência máximo
$PEV(bus,VE,t)$	Potência ativa de carregamento do VE (kWh)

PEV_{mx}	Potência máxima de carregamento do VE (kWh)
PEV	Variação de potência de carga do VE (kWh)
PEV_{mx}	Variação máxima de potência de carga do VE (kWh)
$EV(VE, t)$	Carga do VE no instante de tempo (kW)
$EV_{nom}(VE)$	Carga nominal do VE (kW)
η	Eficiência de carregamento
t_{conn}	Instante de tempo de conexão do VE à rede elétrica (h)
$Si(VE)$	Estado inicial de carga do VE

SUMÁRIO

Capítulo 1

Introdução

Segundo (YAN et al., 2019; WANG et al., 2018), nos últimos anos, houve um aumento no interesse em produzir energia elétrica advinda dos geradores distribuídos (GDs) do tipo eólico e solar, de forma a reduzir os custos de grandes centrais de geração convencionais, obtendo uma maior confiabilidade e eficiência no sistema. Além disso, o veículo elétrico (VE) é apresentado como uma alternativa dos veículos a combustível fóssil, uma vez que utiliza da energia elétrica para sua locomoção, em vez da queima de derivados do petróleo.

Considerando as alternativas mencionadas para solucionar o problema das emissões de gases poluidores, o aumento dos VEs e de GDs na rede distribuição podem ocasionar o surgimento de problemas relacionados à rede elétrica. De acordo com o estudo de (CHEN et al., 2018), é previsto que o aumento de VEs em uma determinada área possa alterar o perfil de consumo e, possivelmente, gerar um grande impacto na rede elétrica, especialmente em horários de ponta ou em determinadas estações do ano.

Em (HAJFOROOSH et al., 2015), o autor cita que é necessário propor técnicas de coordenação de carregamento de VEs, que sejam compreensíveis e aplicáveis ao mercado de energia. Dentre esses algoritmos, são citado os Algoritmos Genéticos Fuzzy e Otimização de Enxame de Partículas Discretas Difusas para carregar as baterias dos VEs. Em ambas as pesquisas citadas previamente, a coordenação do carregamento de VEs mostra uma redução no custo total e nas perdas do sistema. Isso se deve pelo fato que, com o controle descentralizado, ou seja, a critério de cada consumidor controlar os horários de carga, pode-se ter um resultado não ótimo. Um dos exemplos de controle descentralizado, é deixar esse controle de carga com o próprio consumidor, que identifica os melhores horários para a carga do VE, com valores tarifários diferentes de energia em determinadas horas. Esse incentivo de diferentes valores, resulta em menores impactos na rede, mas não garante o carregamento ótimo (CHEN et al., 2018).

Em conformidade com os métodos e conceitos previamente citados, os trabalhos de (GARCIA-OSORIO et al., 2013; ALVAREZ et al., 2018) apresentam soluções para problemas de coordenação de carregamento de VEs, sendo que utilizam técnicas de otimização linear e da Colônia de Abelhas Artificial Aprimorada, respectivamente. Em (ALVAREZ et al., 2018), são apresentadas duas principais estratégias para o gerenciamento de carga dos VEs. A primeira é a descentralizada ou distribuída, que fornece

grande flexibilidade para o proprietário do VE, permitindo a escolha do tempo e maneira de como esse veículo pode ser carregado. Entretanto, essa abordagem pode levar a problemas de sobrecarga na rede elétrica e dificilmente o custo do sistema seria ótimo. A outra estratégia apresentada em (ALVAREZ et al., 2018), é a forma centralizada, em que uma central toma as decisões relacionadas ao planejamento de carregamento dos VEs de uma determinada área. Essa opção é mais recomendada em relação à primeira, em grande parte das pesquisas atuais.

O estudo de (SILVA; RUEDA-MEDINA, 2020) propõe uma solução utilizando uma otimização híbrida para alocação de estações de carregamento de VEs e dos GDs síncronos, em que são utilizados os AG para realizar a alocação e o método dos Pontos Interiores para solucionar o problema de despacho econômico de energia dos GDs. No trabalho de (GARCIA-OSORIO et al., 2013), são utilizados métodos de programação linear para solucionar os possíveis problemas relacionados ao carregamento dos VEs de forma descontrolada.

A pesquisa de (HAJFOROOSH et al., 2015) propõe dois métodos meta-heurísticos, Algoritmos Genéticos *Fuzzy* e Enxame de Partículas Discretas Difusas, para minimizar os custos associados ao fornecimento de energia e as perdas da rede, enquanto maximiza a potência fornecida para os VEs. Contudo, o autor considera que a taxa de carregamento dos VEs é constante, o que pode gerar uma diminuição da vida útil das baterias precocemente, em caso de valores elevados de carga.

Considerando tais informações, o objetivo desta pesquisa foi propor uma solução para otimizar o custo do sistema elétrico de distribuição a partir da coordenação do carregamento de VEs e da geração de potência ativa e reativa dos GDs presentes na rede. Os métodos de otimização utilizados nessa pesquisa são o de AG e o método de Pontos Interiores. Em conjunto a esses métodos, foi apresentada a técnica de *Hardware-in-the-loop* (HIL) em conjunto com um *software* de *High-Level Synthesis* (HLS) aplicados a um AG, de forma a demonstrar o ganho da aceleração de *hardware* no processo de otimização.

O sistema de teste utilizado é uma parte do sistema radial *Institute of Electrical and Electronics Engineers* (IEEE) 37 nós, que consiste nas 15 primeiras barras do sistema. Os detalhes do sistema original IEEE 37 nós podem ser encontrados em (BARAN; WU, 1989). Para resolver o problema de fluxo de potência, foi utilizada a ferramenta *A Mathematical Programming Language* (AMPL), pois apresenta uma forma rápida de modelar problemas de otimização através de equações matemática, sendo essa linguagem de alto nível (FOURER et al., 1990). O AMPL é um sistema integrador que une o modelo matemático, os dados e um pacote de otimização, entregando um resultado ótimo. O pacote de otimização utilizado nesse estudo foi o *Interior Point Optimizer* (IPOPT) para resolver as igualdades do fluxo de potência. O Método dos Pontos Interiores é apresentado em

(LESAJA, 2009).

Esse trabalho possui o diferencial de utilizar tanto o despacho econômico dos GDs e de obter a potência de carregamento otimizado dos VEs a cada hora em um período de 24 horas, utilizando os AG, enquanto que o AMPL é utilizado para resolução do fluxo de potência, fazendo com que o tempo executando o algoritmo seja reduzido. Além disso, também é apresentada a técnica de HIL, podendo reduzir ainda mais o tempo para realizar a otimização.

Nessa pesquisa, foram utilizados 4 casos da rede de distribuição, analisando-se os custos totais e as perdas a cada hora, em um período de 24 horas.

No capítulo seguinte, é apresentada a fundamentação teórica para a compreensão da pesquisa, apresentando os conceitos de carregamento de VEs, dos AG, da simulação em tempo real, do FPGA e dos conceitos principais da programação em paralelo. Em seguida, é apresentada a metodologia utilizada para formular e solucionar o problema de coordenação de VEs em uma rede de distribuição, além de apresentar as propostas de simulação para alcançar o paralelismo de uma das funções dos AG. Por fim, são apresentados os resultados da coordenação dos VEs e das simulações, avaliando os resultados e propondo novas abordagens para aplicar o paralelismo de forma mais efetiva no AG.

1.1 OBJETIVOS

Os objetivos desta Pesquisa são apresentados a seguir, separados entre o Objetivo Geral e Objetivos Específicos.

1.1.1 Gerais

O objetivo desse projeto consiste em propor uma solução otimizada de controle centralizado do carregamento dos VEs, visando reduzir os custos do sistema, utilizando o método de AG, juntamente com uma simulação em tempo real, através de um FPGA e a técnica de *Hardware-in-the-loop*.

1.1.2 Específicos

- Desenvolver o modelo de otimização do sistema de carregamento das baterias dos VEs.
- Aplicar as técnicas de *Hardware-in-the-loop* e simulação em tempo real ao modelo desenvolvido.

- Avaliar os custos dos sistemas sem otimização e otimizado, com objetivo de identificar as vantagens tanto do processo de otimização do carregamento dos VEs, quanto do uso da simulação em tempo real.

1.2 ORGANIZAÇÃO DO TRABALHO

Esta pesquisa é dividida em 5 Capítulos, que são Introdução, Fundamentação Teórica, Metodologia, Resultados e Discussões e Conclusões. O Capítulo 1, Introdução, constitui a apresentação do trabalho e esclarecendo os objetivos que o trabalho se propõe a atingir. Em seguida, no Capítulo 2, é apresentado a fundamentação teórica dos temas abordados na pesquisa, além do estado da arte sobre a coordenação de carregamento de VEs utilizando a técnica de HIL e o FPGA. No Capítulo 3, são exibidas as propostas de simulações e testes para atingir os objetivos previamente apresentados, além dos dados do sistemas de teste. No Capítulo 4 são apresentados os resultados e discussões do testes e simulações realizados. Por fim, no Capítulo 5 são retomados os objetivos desta pesquisa para apresentar os que foram alcançados e os que não foram, sugerindo temas e métodos para futuras pesquisas.

1.3 PUBLICAÇÕES

A partir de resultados parciais desta pesquisa, foi possível publicar um artigo no congresso nacional IX Simpósio Brasileiro de Sistemas Elétricos (SBSE 2022), com título Coordenação e Otimização de Carregamento de Veículos Elétricos e Despacho Econômico de Geradores Distribuídos Utilizando Algoritmos Genéticos.

Capítulo 2

Fundamentação Teórica

2.1 FLUXO DE POTÊNCIA

Segundo (ZHU, 2015), o estudo do fluxo de potência de um sistema de potência diz respeito à solução que mostra os valores de corrente, tensão, potência ativa e potência reativa em todos os nós dentro do sistema. Considerando que os parâmetros de uma rede são constantes, como transformadores e linha de distribuição ou transmissão, a rede é considerada linear, entretanto, como as relações de tensão, corrente, potência ativa e potência reativa são não lineares. Assim, o cálculo do fluxo de potência envolve equações não lineares, de forma a obter as respostas elétricas de um sistema de transmissão ou distribuição.

Ainda em (ZHU, 2015), são demonstradas as equações utilizadas para solucionar o problema de fluxo de potência. Geralmente, uma rede com n nós independentes pode ser descrita em n equações, como em na Equação 1.

$$\begin{cases} Y_{11}\dot{V}_1 + Y_{12}\dot{V}_2 + \cdots + Y_{1n}\dot{V}_n = \dot{I}_1 \\ Y_{21}\dot{V}_1 + Y_{22}\dot{V}_2 + \cdots + Y_{2n}\dot{V}_n = \dot{I}_2 \\ \vdots \\ Y_{n1}\dot{V}_1 + Y_{n2}\dot{V}_2 + \cdots + Y_{nn}\dot{V}_n = \dot{I}_n \end{cases} \tag{1}$$

Onde, o Y_{ij} é o negativo da admitância do ramo entre os nós, enquanto que a diagonal Y_{ii} é denominada auto admitância, que é igual a soma de todas as admitâncias ligadas ao nó. Os termos V e I são respectivamente a tensão nos nós e a corrente de injeção dos nós.

2.1.1 Fluxo de potência ótimo

O fluxo de potência ótimo (FPO) foi introduzido por Carpentier em (CARPENTIER, 1962). O objetivo do FPO é encontrar uma configuração de um sistema de energia de forma a otimizar a função objetivo do sistema, como custo de geração, custo total, perdas do sistema, entre outros. O sistema é definido a partir de equações lineares ou não lineares, que dizem respeito às variáveis do sistema e dos limite de operação dos equipamentos e do sistema

De acordo com (ZHU, 2015), existem diferentes tipos de formulações matemáticas para diferentes função objetivo e restrições para um problema de FPO. As principais classificações são:

- Problemas lineares: possuem função objetivo e restrições na forma de variáveis de controle contínuas e lineares;
- Problemas não lineares: consistem em uma função objetivo ou restrições ou ambos com variáveis de controle contínuas e não-lineares;
- Problemas lineares com inteiros: chamados de *Mixed-integer linear problems* (MILP), esse problema possui variáveis de controle contínuas e inteiras.

Dessa forma, foram criadas diversas técnicas para solucionar tais problemas como, por exemplo, técnicas clássicas, como o Método de Newton, que requer a utilização de derivadas parciais de segunda ordem das equações de fluxo de potência. A técnica clássica Método de Pontos Interiores (MPI) é outro exemplo, que foi originalmente proposto para resolver problemas lineares, mas foi melhorado e atualmente é utilizado para solucionar problemas de FPO. Outra técnica utilizada para resolução de problemas não-lineares, é o método de Gauss-Seidel, que também é utilizado para solucionar equações de fluxo de potência.

Além das técnicas clássicas, também foram criadas técnicas meta heurísticas, como os AG, que consiste em aplicar a teoria evolutiva de Darwin de forma a obter melhores resultados a partir da combinação de boas respostas, com a vantagem de não necessitar de aplicar operações computacionalmente custosas, como aplicação de gradiente. No escopo de técnicas meta heurísticas, existem diversas outras técnicas, como a Busca Tabu, que também é um método iterativo que tem como característica a capacidade de evitar mínimos locais dentro do espaço de busca. Outro método meta heurístico utilizado é a Otimização de Enxame de Partículas, que é inspirado no comportamento de colônias e exploram regiões promissoras dentro do espaço de busca. Além de todos esses métodos ditados, existem diversas outras técnicas que são detalhadas em (ZHU, 2015).

2.2 VEÍCULOS ELÉTRICOS

Atualmente, o número de VEs vem crescendo a cada ano. A escolha de um VE, incentivada por governos, mostra uma preocupação com a redução de gases poluentes advindos de veículos que utilizam combustíveis fósseis (BOULANGER et al., 2011; BALEN et al., 2019). Entretanto, os VEs possuem algumas desvantagens como, por exemplo, sua autonomia, que geralmente são inferiores aos veículos à combustão, além do tempo de carregamento do VE ser maior. Dessa forma, existem opções de VEs no mercado, que tentam mitigar essas desvantagens. Os principais tipos de VEs são:

- Veículo Elétrico Híbrido (HEV)
- Veículo Elétrico Híbrido *Plug-in* (PHEV)
- Veículo Elétrico *Plug-in* (PEV)

2.2.1 Tipos de Veículos Elétricos

Os HEVs tem como característica o seu abastecimento igual ao veículo à combustão convencional, entretanto, esses possuem um sistema de baterias que contribui em sua eficiência, tendo um consumo de combustível (gasolina, etanol ou diesel) inferior aos convencionais. Dessa forma, sua autonomia não possui impactos consideráveis, porém existe apenas uma pequena redução na emissão de gases.

Os PHEVs, assim como os HEVs, consomem, majoritariamente o combustível tradicional. A diferença desse VE é que é possível conectá-los à rede elétrica para recarga das baterias, e por isso, esse modelo possui um banco de baterias maior, refletindo em um maior eficiência energética e menor emissão de gases poluentes.

Por fim, os PEVs são VEs que não consomem combustível fóssil diretamente e sua fonte de energia é totalmente elétrica. Por esso motivo, esse é o modelo que menos emite gases poluentes, porém geralmente possui uma autonomia menor. Esses veículos podem ser carregados em postos de abastecimento elétrico ou ainda dentro da própria residência, à depender do tipo de tomada que é utilizada. Além da baixa autonomia, a velocidade de carregamento é baixa, mesmo em casos de carregamentos rápidos.

2.2.2 Tipo de Carregamento

O primeiro tipo de carregamento é chamado de Carregamento Nível 1, que é o tipo de carregamento mais comum e mais utilizado, pois pode ser feito dentro de casa. Esse carregamento é comumente chamado de carregamento lento ou tipo 1. O próximo tipo de carregamento é o Carregamento Nível 2, também chamado de carregamento semi-rápido. Esse carregamento é utilizado em instalações privadas ou públicas e requer uma tensão maior. Por fim, o Carregamento Nível 3, ou ainda chamado de carregamento rápido, é utilizado comercialmente em postos de abastecimento para um rápido abastecimento. Esse tipo de carregamento requer equipamentos com uma proteção maior e, consequentemente, são mais caros, por isso é mais utilizado comercialmente (YILMAZ; KREIN, 2012). A Tabela 1 mostra, de forma resumida, os tipos de carregamento e seus níveis de tensão (à depender da região).

Tabela 1 – Tipos de carregamentos de VE com *plug-in*

Tipos	Uso típico	Potência esperada	Tempo de carregamento	Tecnologia
Nível 1 230 V	Residencial	1,9 kW (20A)	11 a 36 horas	PHEVs e PEVs
Nível 2 400 V	Estacionamentos	19,2 kW (80A)	2 a 3 horas	PHEVs e PEVs
Nível 3 600 V	Postos de abastecimento	100 kW	0,2 a 0,5 horas	Apenas PEVs

Fonte: Traduzido e adaptado de (YILMAZ; KREIN, 2012).

2.3 ALGORITMOS GENÉTICOS

Os AG são um método da computação evolutiva que toma como base a teoria evolutiva de Charles Darwin. Esse método foi desenvolvido por John Holland durante os anos 1960 e 1970 e se popularizou por meio de um de seus estudantes, David Goldberg (HAUPT; HAUPT, 2004), que conseguiu solucionar problemas difíceis envolvendo dutos de transmissão de gás em sua dissertação.

Esse método busca, em sua população de soluções, os indivíduos mais bem adaptados ao meio, ou seja, com o melhor valor obtido através de suas funções de avaliação (*fitness function*). A partir desses melhores resultados, esses indivíduos são classificados, sendo realizada seleção natural, descartando os indivíduos menos adaptados e mantendo os melhores. Em seguida, os melhores indivíduos são separados em pares para que haja o cruzamento (*crossover*) dessas soluções de forma a gerar melhores indivíduos (WANG et al., 2018).

Após o cruzamento, ocorre a mutação, gerando assim variações aleatórias nos indivíduos. Assim, todos esses indivíduos compõem a nova população de soluções. De acordo com (WANG et al., 2018), os AG não necessariamente fornecem a solução ótima para o problema, mas apresentam uma solução de ótima qualidade.

É válido ressaltar que a escolha dos parâmetros supracitados é de extrema importância para que o algoritmo tenha uma bom desempenho, pois escolhas realizadas sem critério podem gerar algoritmos pouco performáticos ou que não convergem, devido a alta ou baixa aleatoriedade do mesmo.

Algumas das vantagens de se utilizar os AG, segundo (HAUPT; HAUPT, 2004) são:

- Otimização de variáveis contínuas ou discretas,
- Não necessita de informações de derivadas,

- Busca simultaneamente dentro de uma grande quantidade de amostras dentro de uma superfície de custo,
- Consegue processar com uma quantidade grande de variáveis,
- É adequada para processamento em paralelo,
- Otimiza variáveis em uma superfície de custo extremamente complexa,
- Provê uma lista de variáveis ótimas, não apenas uma solução.

Uma grande desvantagem dos AG, é a sua velocidade de processamento que geralmente é inferior aos métodos de otimização clássica (HAUPT; HAUPT, 2004). Entretanto, muitas vezes, os métodos clássicos não são capazes de encontrar soluções que possuem um grau de complexidade mais elevado, sendo necessário a utilização de algoritmos meta-heurísticos ou heurísticos, como por exemplo, os AG. Em (HAUPT; HAUPT, 2004), é afirmado que o verdadeiro poder de otimização dos AG é quando esse algoritmo é utilizado com um computador de processamento em paralelo, uma vez que as populações podem ser processadas de forma independente, calculando todas as funções *fitness* simultaneamente.

2.4 AMPL

O *software* AMPL ou *A Mathematical Programming Language* foi proposto e desenvolvido por Robert Fourer, David Gay e Brian Kernighan dentro do Laboratório Bell, por volta do ano 1985, para ajudar as pessoas a desenvolver e implementar problemas práticos de otimização, usando modelos matemáticos (FOURER et al., 1990). A linguagem utilizada no *software* também é denomidada AMPL e a principal proposta dos autores com essa tecnologia é reduzir o tempo de programação em si, focando majoritariamente o tempo dos desenvolvedores na modelagem do problema.

A sequência completa de eventos para solucionar problemas matemáticos de otimização é dado pelo passo a passo (FOURER et al., 1990):

- Formular o modelo, o sistema de variáveis já abstraídos, objetivos e restrições que representam o modelo a ser resolvido.
- Coletar dados que definem uma instância do problema.
- Gerar uma função objetivo específica e as equações de restrições a partir do modelo e dos dados.
- Resolver a instância do problema rodando o programa, ou *solver*, para aplicar o algoritmo que encontra a solução ótima para as variáveis.

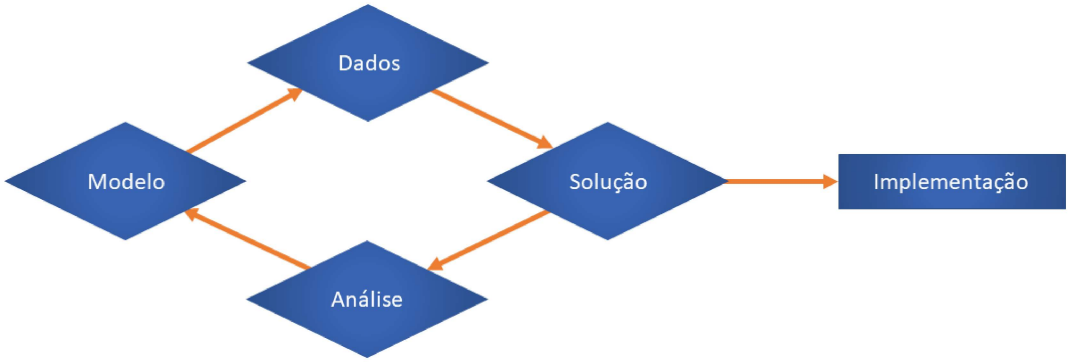
- Analisar os resultados.
- Refinar o modelo e os dados conforme o necessário e então, repetir o processo.

Entretanto, para que seja possível solucionar problemas computacionais, é necessário que esse modelo proposto seja transcrito para algum tipo de algoritmo, de forma a ser possível executá-los no computador. Nesse caso, muitas das vezes esse processo de transcrição consome muito tempo, é custoso e gera uma grande chance de erro humano na transcrição, dispendendo mais tempo ainda para encontrar o erro.

O AMPL oferece um ambiente interativo de comandos para organizar o modelo e os dados do problema matemático de otimização, para que em seguida seja solucionado. Dessa forma, o passo a passo descrito anteriormente possa ser aplicado quase diretamente, resultando em um maior período de modelagem e não de transcrição. O *software* também possibilita a troca de pacotes de otimização de forma transparente, tendo poucas alterações no código e permitindo que o desenvolvedor possa testar os melhores *solvers* para o seu problema, sem dispender de um tempo muito grande.

O processo de utilização do AMPL é mostrado na Figura 1. Nessa figura é possível ver que o processo de otimização de um processo começa com a criação do modelo e obtenção de dados para em seguida ser solucionado pelo *solver*. Após esse processo, os resultados são analisados e são feitas adaptações no modelo e inserido mais dados, resultando num ciclo indefinido, até que o modelo esteja com resultados satisfatórios, quando pode ser realizada a implementação. Vale ressaltar que o processo pode sempre voltar ao ciclo de modelagem, caso seja necessário manutenção ou inserção de novos dados.

Figura 1 – Diagrama do processo de utilização do *software* AMPL



Fonte: Próprio autor

Como pode ser visto, é necessário adquirir um pacote de otimização, pois o AMPL fornece uma solução de apenas integração e não de otimização em si. Existem diversos pacotes de otimização no mercado, tanto pagos como gratuitos, e cada um propõe solucionar uma gama definida de problemas, como lineares, lineares com inteiros misto, não lineares, entre outros. Para o problema dessa pesquisa, optou-se pelo IPOPT, que é um pacote de otimização gratuito e que soluciona problemas não lineares contínuos, fornecendo uma otimização de mínimo ou máximo local, utilizando o MPI.

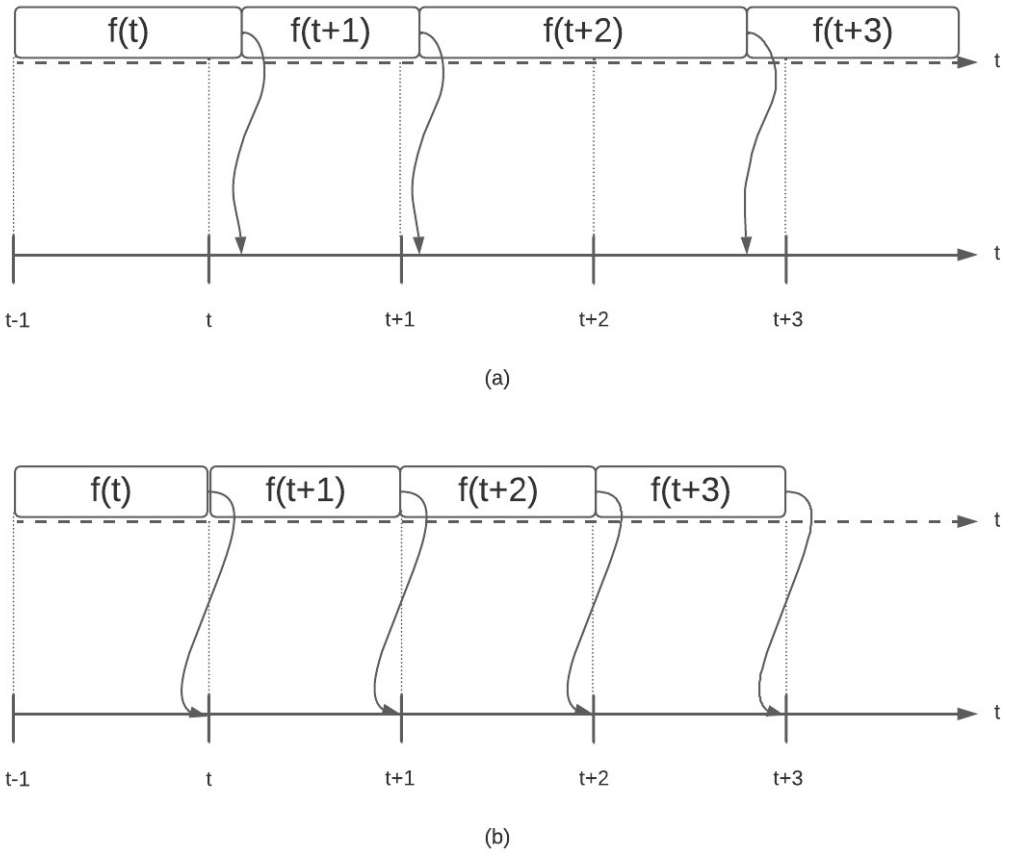
O MPI que é descrito em (LESAJA, 2009) como sendo um algoritmo muito utilizado em pesquisas de otimização, uma vez que possui uma velocidade de convergência relativamente rápida. Esse método foi inicialmente proposto no trabalho de (KARMARKAR, 1984) como sendo um algoritmo promissor para usos práticos. Diferente dos algoritmos até então propostos, como o método simplex, que calcula as iterações nas regiões de contorno, o MPI realiza as iterações no interior da região factível. Para aplicar esse método, o problema original deve ser transformado em uma forma especial, aplicando transformações projetivas e mapeando a iteração atual no centro do conjunto especial, onde se é calculada o mínimo da função e em seguida, é mapeada de volta para o espaço original. A descrição mais detalhada do MPI pode ser encontrado no trabalho de (WACHTER, 2002).

2.5 SIMULAÇÃO EM TEMPO REAL

A simulação é um ensaio que reproduz, de forma artificial, uma situação real considerando as condições de um meio, através de um modelo. A simulação em tempo real consiste em obter sucessivamente a solução do modelo dentro de um passo de tempo, previamente definido (BÉLANGER et al., 2010; FARUQUE et al., 2015). Caso o tempo para a obtenção do resultado do modelo for superior ao passo de tempo estipulado, a simulação é considerada uma simulação *offline*. A simulação em tempo real pode ser utilizada em sistemas de potência para simular os transientes da rede, ou ainda para

obter as soluções de modelos com um grande número variáveis e parâmetros (BÉLANGER et al., 2010) que necessitam de uma solução do modelo com maior velocidade e maior fidelidade aos resultados reais, se tornando uma boa opção para realizar a prototipagem. Na Figura 2 são apresentados os modelos de simulação *offline* (a) e simulação em tempo real (b).

Figura 2 – (a) Simulação *offline*, (b) Simulação em tempo real



Fonte: Próprio autor.

Segundo (FARUQUE et al., 2015), os simuladores de tempo real aplicados em sistemas de potência podem ser classificados em duas categorias, a simulação totalmente digital em tempo real e a simulação com HIL. Na primeira categoria, são utilizadas as técnicas de *Model-in-the-loop* ou *Software-in-the-loop*, que necessitam da modelagem de todos os componentes do sistema dentro do próprio simulador, porém não possui interfaces de entrada ou saída. Na categoria de simulação com HIL, parte do sistema digital da simulação são transferidos para componentes físicos, que se comunicam com o modelo digital através de interfaces de entradas e saídas. Em geral, a utilização de ambos varia conforme a necessidade, porém os sistemas totalmente digitais são geralmente utilizados para o entendimento do funcionamento do sistema para certas circunstâncias;

enquanto o HIL é utilizado para diminuir o risco de investimento, através de um protótipo ou quando não se é possível utilizar um sistema real.

Atualmente, existem diversas pesquisas como, por exemplo, (ROINILA et al., 2018; WANG et al., 2019; SINGH et al., 2020) que aplicam o conceito de simulação em tempo real e HIL em sistemas elétricos de potência. Em (ROINILA et al., 2018), é apresentada uma pesquisa que utiliza da técnica do HIL, baseada no simulador em tempo real da OPAL-RT para modificar características do sistema em tempo real como, por exemplo, o comportamento da impedância e dinâmica de controle, sendo possível prover uma solução estável e adaptativas para um sistema de distribuição. Em (WANG et al., 2019), foi utilizado a técnica de HIL para controlar a tensão através de um inversor, deixando as mudanças de tensão mais suaves, restringindo os problema de elevações e quedas, e ainda coordenando, em tempo real, painéis fotovoltaicos. A pesquisa de (SINGH et al., 2020), tem como objetivo estimar o estado de carga (SoC) de uma bateria de íon de lítio através de seu modelo proposto. Para que sejam acompanhados os valores de todos os parâmetros do modelo, o autor utilizou a técnica de HIL e simulação em tempo real.

Dessa forma, é possível verificar que estudos recentes apontam para a utilização de simulação em tempo real e a técnica de HIL, de forma a deixar as predições de um sistema mais assertivas.

Por fim, para implementar a técnica de HIL à simulação em tempo real, (RAZZAGHI et al., 2013; MEDINA, 2019; MORELLO et al., 2018) apresentam o uso do *Field Programmable Gate Array* (FPGA), que possibilita a implementação em paralela do algoritmo, o que reduz drasticamente a sequência de operações em relação a um computador para aplicações genéricas. Essa vantagem resulta em uma diminuição do mínimo passo de tempo para solucionar o modelo. Em (MEDINA, 2019), também é detalhado que esse aumento de velocidade de processamento decorre da exclusividade de execução de tarefas do *hardware* e apresenta o projeto FAPES/CNPq 141/2017 denominado de SIMUTERE, que consiste em incluir o FPGA dentro do laço de repetição de controle do sistema.

2.6 FPGA

O FPGA é um dispositivo formado por semicondutores, que são baseados em um matriz configurável de blocos lógicos (XILINX, 2022b). Os FPGAs podem ser reprogramados para atender as necessidades do usuário em relação a funcionalidade ou aplicação desejável, mesmo depois de sua confecção. Existem dois tipos de FPGA, os *one-time programmable* (ou OTP) FPGAs e os FPGAs baseados em *Static Random Acess Memory* (ou SRAM), sendo possível, nos respectivos tipos, serem programados apenas uma vez e

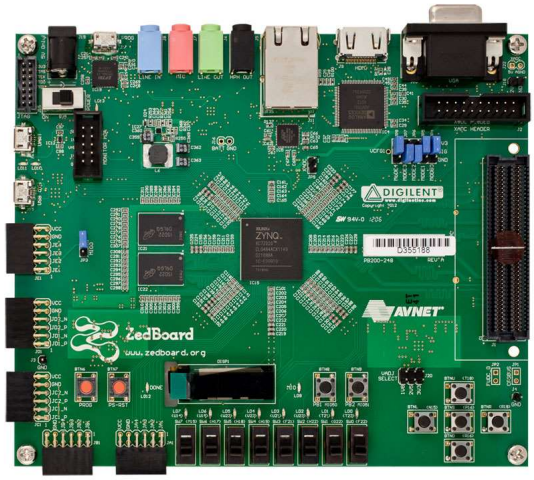
serem reprogramáveis diversas vezes. De acordo com o fabricante, os tipos que dominam o mercado são o do tipo SRAM, que podem ser readequados a novas configurações, a medida que o projeto evolui (XILINX, 2022b).

Por serem reprogramáveis, os FPGAs são muito flexíveis e podem ser utilizados em diversos mercados. O fabricante ressalta alguns mercados que o FPGA participa, existindo diferentes modelos para cada cenário. Alguns dos mercados são:

- *Data Center*
- Processamento computacional de alto desempenho
- Processamento de imagem
- Aplicações industriais
- Prototipagem

A programação do FPGA pode ser feita em duas linguagens: Verilog ou *Very High Speed Integrated Circuit Hardware Description Language* (VHDL). Segundo (WILSON, 2015), Verilog é uma linguagem de mais baixo nível, enquanto o VDHL é de mais alto nível, mas ambas as linguagens são utilizadas mundialmente e a escolha delas depende de diversos fatores, como preferência pessoal ou decisões da empresa. Ainda há também a possibilidade de utilizar ambas as linguagens, utilizando bibliotecas pré-compiladas, por exemplo. Entretanto, as duas linguagens são utilizadas para descrição de *hardware* e possuem diferenças entre linguagens de programação amplamente difundidas e um grau de complexidade elevada. A Figura 3 mostra o modelo ZedBoard Zynq-7000.

Figura 3 – Modelo ZedBoard Zynq-7000



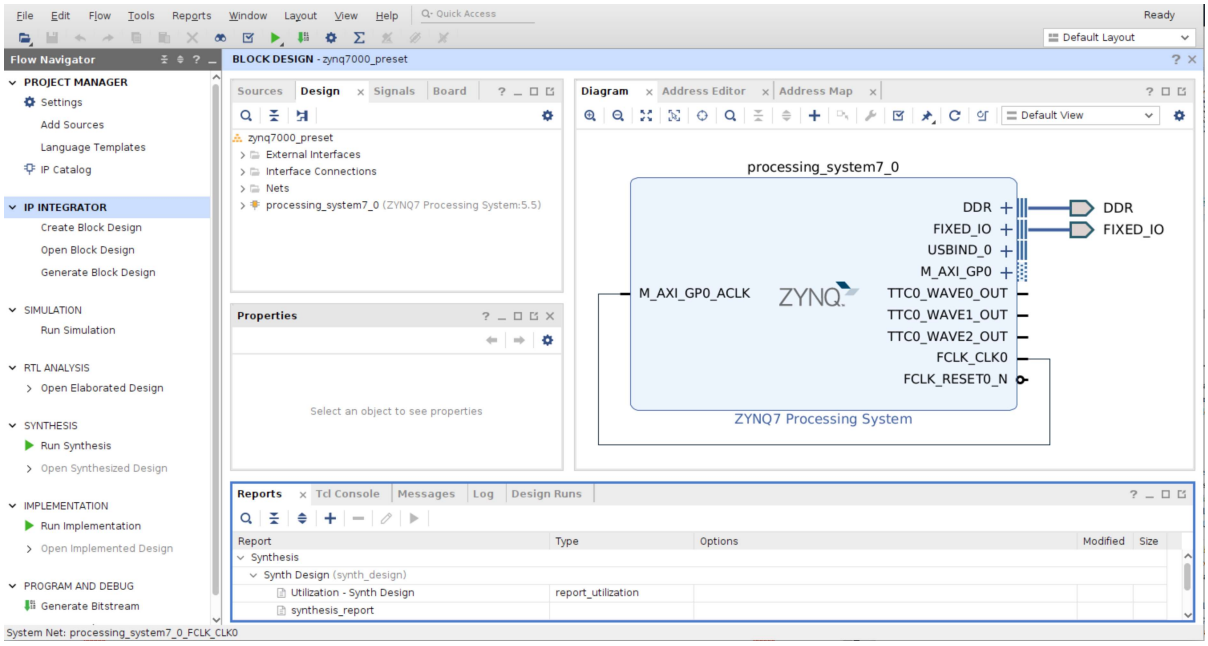
Fonte:(XILINX, 2022b)

2.7 *SOFTWARES* PRODUZIDOS PELA XILINX

2.7.1 Vivado

Para acessar e programar o *hardware*, a empresa fabricante, Xilinx, oferece soluções em *software* que auxiliam no desenvolvimento. O Vivado é a ferramenta proposta pela empresa para elaborar projetos que configurem as portas lógicas do FPGA da forma almejada no projeto. Na Figura 4, é apresentada a interface gráfica do Vivado.

Figura 4 – Interface gráfica do *software* Vivado



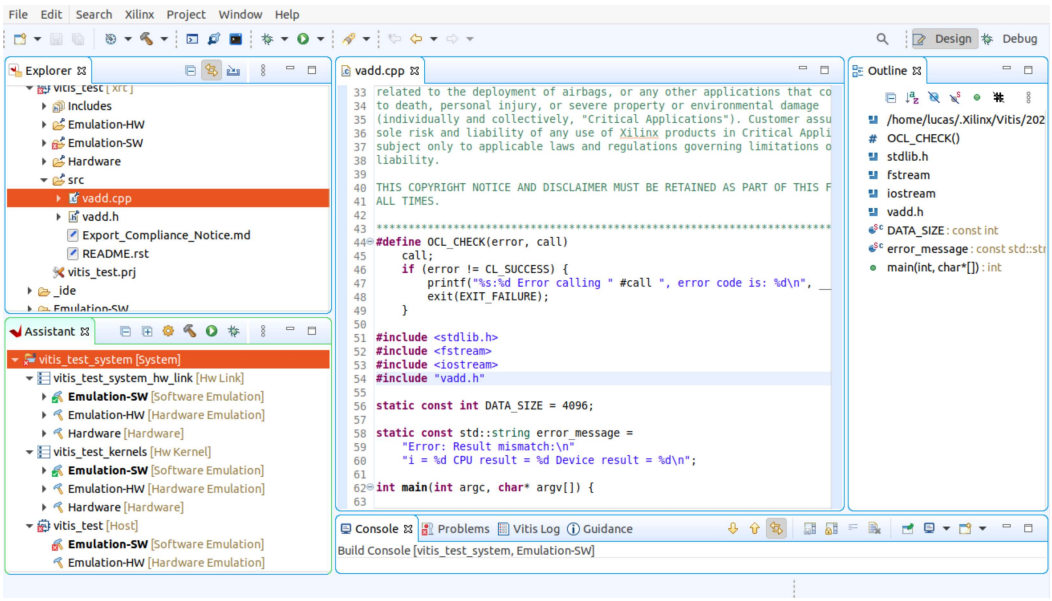
Fonte: (XILINX, 2022b)

Na interface apresentada na Figura 4, é mostrada o espaço de trabalho que o programador normalmente utiliza. Na parte da direita, é apresentada a área de desenvolvimento do diagrama que é implementado no FPGA. À esquerda, aparece o Navegador de Fluxo (*Flow Navigator*), que possui as telas de configuração do seu sistema, além do fluxo do processo para programar o FPGA. O fluxo principal é mostrado nas opções de Síntese (*Synthesis*), que é responsável pela conversão do projeto à nível de registro, que também é chamado de *Register Transfer-Level* (RTL), para a representação em nível de portas lógicas (XILINX, 2019). Em seguida, é realizada a implementação, que é o processo de atribuir a utilização e limitações de recursos que serão utilizadas, como as restrições lógicas, físicas e de tempo do projeto (XILINX, 2019). Por fim, é gerado o arquivo executável e binário que é enviado para o FPGA, na parte de Geração do Fluxo de Bits (*Generate Bitstream*).

2.7.2 Vitis

Além de programar o *hardware*, também há a possibilidade de utilizar o *software* Vitis, que permite o uso de linguagens de nível mais alto para programar, ao invés das linguagens VHDL ou Verilog. O Vitis é uma outra ferramenta fornecida pela Xilinx, que está mais próxima da programação tradicional do que a descrição de *hardware*. Essa ferramenta propõe utilizar outras linguagens, como C e C++, para programar o FPGA, utilizando a HLS, que traduz os comandos de C/C++ em Verilog/VHDL (MEDINA, 2019). Na Figura 5, é apresentada a interface gráfica do Vitis.

Figura 5 – Interface gráfica do *software* Vitis



Fonte: (XILINX, 2022b)

A interface gráfica apresentada na Figura 5 possui menos componentes na área de trabalho. À direita, é possível ver a tela que apresenta as variáveis e seus estados durante a execução do código, enquanto que ao centro é apresentado o código para ser editado. À esquerda, aparece o Explorador de Arquivos do Projeto (*Explorer*), que mostra todos os arquivos presentes no projeto, enquanto que logo abaixo, apresenta o Assistente de *Software* (*Assistant*), que mostra os arquivos compiláveis e a se a compilação possui erros ou está correta.

Apesar da ferramenta reduzir o esforço e conhecimento das linguagens descritivas de *hardware*, ainda é necessário desenvolver a interface entre a comunicação entre o computador e o FPGA, habilitando o processamento em paralelo dos dados, que é responsável por habilitar velocidades altas de processamento.

2.8 COMPUTAÇÃO PARALELA

Geralmente, o FPGA era selecionado por sua baixa velocidade (cerca de 50 a 100 MHz), complexidade e volume, mas atualmente alguns FPGA conseguem passar a barreira de 500 MHz (XILINX, 2022b). Mesmo com as velocidades atuais, se comparado aos processadores dos computadores atuais (por volta de 3 GHz), ainda possuem uma frequência operacional baixa. Entretanto, as *Central Processor Units* (CPUs), ou unidades de processamento central, conseguem apenas processar instruções em série e algumas poucas instruções em paralelo (a depender de quantos núcleos ele possui). Mesmo assim, é esperado que a velocidade de processamento da CPU seja alta, por conta de sua alta frequência de operação e por trabalhar em conjunto com a memória *cache* (ASANO et al., 2009).

Os motivos do FPGA possuir um alto desempenho vem da sua flexibilidade, que possibilita utilizar seus circuitos de forma dedicada para cada aplicação, além de um número elevado bancos de memória, o que possibilita realizar um alto número de instruções de forma paralela (ASANO et al., 2009). Em relação à Unidade de Processamento Gráfico (GPU), que também realiza operações em paralelo, o FPGA ainda possui maior desempenho em certas aplicações, pois a GPU tem seus núcleos agrupados e a transferência de dados entre os grupos é muito limitada (HANGÜN, 2021).

Por esse motivo, essa pesquisa propõe utilizar o FPGA para processar paralelamente os cromossomos de cada população, de forma a conseguir um desempenho superior e conseguir realizar a simulação em tempo real.

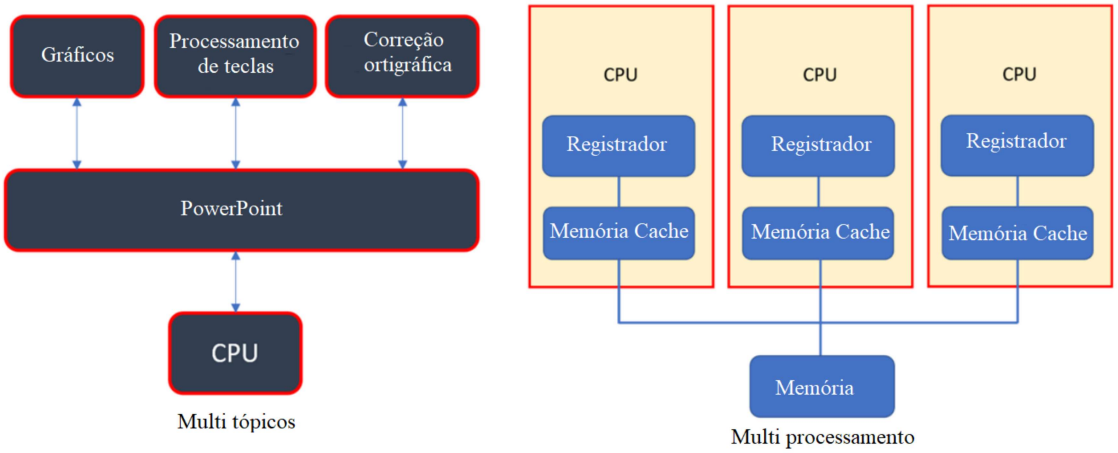
2.9 ARQUITETURA DOS COMPUTADORES TRADICIONAIS

Apesar do FPGA produzir resultados mais rápidos e com menor consumo energético, surge a dúvida sobre o motivo pelo qual a programação em paralela não é amplamente utilizada em uma gama maior de soluções. Para que seja identificada a resposta, é necessário entender como um programa ou *software* funciona em uma máquina tradicional. A Máquina de Von Neumann foi um modelo proposto há mais de 7 décadas e essa arquitetura foi considerada ótima para uma vasta classe de aplicações por ser flexível e programável (XILINX, 2022a). Essa arquitetura faz com que a máquina realize operações da forma que são escritas, ou seja, de forma sequencial, porém com o avanço da tecnologia, foi necessário que parte desses processos ocorressem em paralelo, fazendo que uma CPU realizasse mais de uma operação simultaneamente, surgindo assim os conceitos de *Multithreading* (ou multi tópicos) e *Multiprocessing* (ou multi processamento).

2.9.1 *Multithreading* e *Multiprocessing*

O *Multithreading* é a capacidade de um programa ou processo de realizar diversas operações em paralelo, dentro das limitações de uma CPU, executando diversas operações ou elementos dentro de um *software*. Um exemplo dado por (XILINX, 2022a) é o PowerPoint, que executa operações como processamento do gráfico exibido na tela, pressionamento de teclas e correção ortográfica, utilizando múltiplos *threads* (ou tópicos) em paralelo. Além do processamento de *threads* em paralelo, foi necessário ainda utilizar a técnica de *Multipleprocessing*, que consiste em utilizar diversos núcleos de uma CPU para realizar múltiplos processos dentro de uma mesma máquina. Dessa forma, como mostra a Figura 6, a técnica de *Multipleprocessing* opera com diversas CPUs, nas quais podem ou não operar com a técnica de *Multithreading* para cada processo.

Figura 6 – *Multithreading* e *Multiprocessing*



Fonte: Traduzido de (XILINX, 2022a).

2.10 PARADIGMAS DA PROGRAMAÇÃO EM PARALELO

Para que seja possível realizar a programação em paralelo, é necessário entender os paradigmas, em especial, do FPGA. Essa mudança de paradigma é necessária, uma vez que muita das linguagens, como o C/C++, foram escritas com base na máquina de Von Neumann, que é essencialmente sequencial. Os três principais paradigmas são o Produtor-Consumidor, *Streaming* e *Pipelining* (XILINX, 2022a).

2.10.1 Produtor-Consumidor

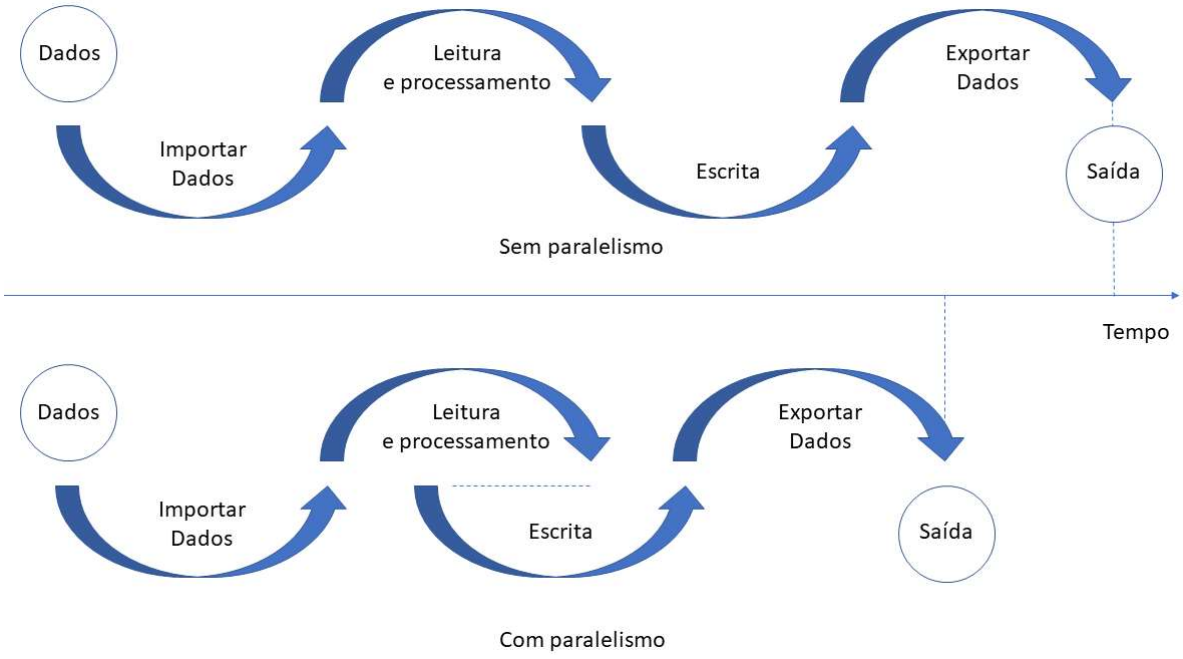
O paradigma do Produtor-Consumidor diz respeito à modelagem de um sistema que consiste em consumir os dados quando estão prontos. Dessa forma, o programador deve ser capaz de identificar as partes que podem ser separadas em atividades em paralelo ou sequenciais, de forma a aumentar o desempenho do sistema. Em um sistema

convencional, escrito em C/C++, primeiro é feita a leitura de todos os elementos de um vetor de dados, para depois serem utilizados ou escritos em outra parte do programa (XILINX, 2022a).

Para exemplificar o uso paradigma Produto-Consumidor, considera-se a seguinte situação: é necessário realizar uma entrada de dados em um programa e, a partir desse dados importados, fazer a leitura e processamento do mesmo, escrevê-los em memória e por fim, exportá-los para fora do programa. Como pode ser visto, existe uma sequência para que todas as tarefas sejam executadas, umas vez que para exportar o dados, é necessário escrever os dados em memória, que por sua vez precisam ser lidos e processados depois de importados.

Entretanto, quando se utiliza o FPGA, não é necessário esperar que todos os dados sejam lidos para começar a escrevê-los e exportá-los. Dessa forma, é possível começar a escrevê-los assim que os primeiros elementos já tiverem sido lidos, diminuindo o tempo total de execução, como mostra a Figura 7.

Figura 7 – Exemplo do paradigma Produtor-Consumidor



Fonte: Próprio autor.

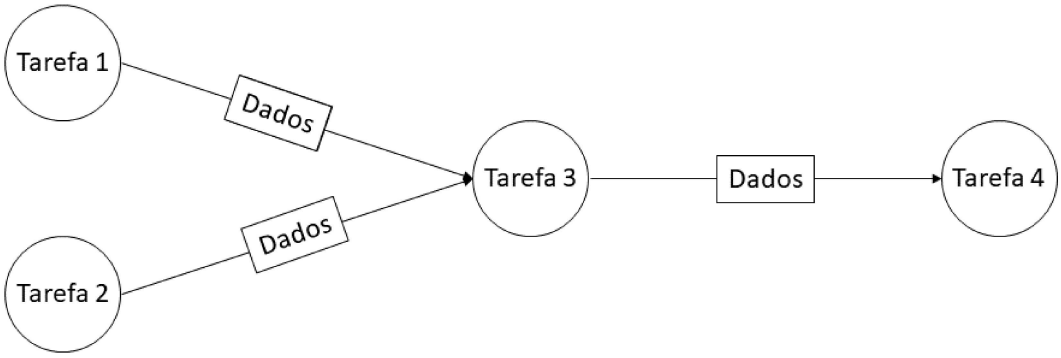
2.10.2 *Streaming* de dados

Segundo (XILINX, 2022a), o conceito de *stream* é importante pois introduz o conceito de comunicação de dados de tamanho ilimitado ou de tamanho desconhecido. Esse conceito está relacionado ao paradigma de produtor consumidor, uma vez que os dados transmitidos pela origem devem ser consumidos pelo destino à medida que são enviados. Isso também permite abstrair algumas complexidades do paralelismo, como

por exemplo o sincronismo de dados entre cada tarefa, permitindo assim um maior desempenho.

Geralmente, esse conceito é aplicado no modelo de *First-In-First-Out* (ou FIFO), que significa que o primeiro dado a entrar no escopo de uma tarefa, é o primeiro a sair para a tarefa subsequente. Aplicando esse método, é possível abstrair o comportamento paralelo do sistema, e consequentemente facilitando sua programação. A Figura 8 mostra graficamente como é um programa utilizando a técnica de *stream*.

Figura 8 – *Streaming*



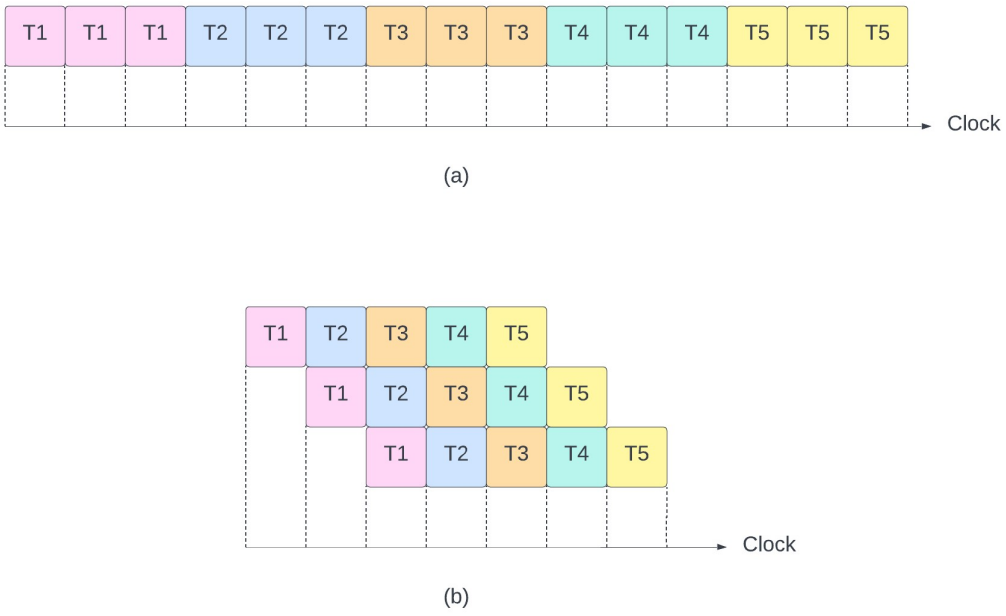
Fonte: Próprio autor.

2.10.3 *Pipelining*

O *Pipelining* diz respeito ao enfileiramento de tarefas que são executadas na máquina. Isso significa que, dentro de um mesmo período de execução, é possível executar uma quantidade limitada de tarefas, até o próximo ciclo. Um ciclo de execução de uma tarefa, também denominado de *clock*, é limitado pelo *hardware* que está executando essas tarefas, sendo mais rápida a execução, quanto maior for o número de *clocks* dentro de um segundo. Entretanto, é possível utilizar múltiplas unidades de processamento, para que essas tarefas possam ser executadas em paralelo, fazendo com que um programa que tem um número fixo de tarefas, possa ser executado em menos tempo em relação a um processamento sequencial.

Para exemplificar essa afirmação, a Figura 9 mostra a diferença da quantidade de *clocks* para executar um código que possui cinco tarefas (na figura denominada T) e que cada uma gasta três ciclos de *clock* para ser finalizada. Cada linha da figura simula um núcleo e nesse caso, a comparação é feita entre uma unidade de processamento de um núcleo em 9(a) e uma unidade de processamento de três núcleo em 9(b). A diferença já é evidente apenas com poucos núcleos, tarefas com poucos ciclos de *clock* e uma quantidade pequena de tarefas.

Figura 9 – (a) Processamento sequencial, (b) Processamento paralelo



Fonte: Próprio autor.

2.11 CONSIDERAÇÕES PARA APLICAR O PARALELISMO

Para paralelizar o código, é possível aplicar os métodos previamente descritos e obter o resultado esperado. Entretanto, para maior aproveitamento dessa técnica, é interessante ter o conhecimento de alguns conceitos de dados para que possa ser possível extrair ao máximo da técnica de paralelismo (XILINX, 2022a).

2.11.1 Taxa de Transferência

Nessa seção, referente aos paradigmas, é apresentado de forma implícita que o tempo de execução está diretamente relacionado com a taxa de transferência. É importante ressaltar que o sistema em paralelo é tão lento quanto sua operação mais demorada (XILINX, 2022a), sendo necessário fazer uma análise detalhada do tempo de execução de cada tarefa e concentrar os esforços no paralelismo das execuções que utilizam maiores ciclos de *clock* para serem concluídas.

Para atingir uma taxa de transferência maior, é necessário que se reduza também, além das tarefas, a latência da transferência de dados. A comunicação entre a CPU e o FPGA é geralmente custosa e para que isso não tenha um impacto grande, pode-se optar por aumentar o espaço utilizado pelos dados, na memória global, fazendo um *trade-off* entre espaço e tempo.

2.11.2 Alinhamento dos dados e Espaçamento entre os dados

Tendo em vista o que foi abordado no tópico anterior, é necessário reduzir ao máximo o tamanho dos dados trafegados, de forma a obter uma resposta mais lenta. Segundo (XILINX, 2022a), em alguns sistemas de 32-bits, o tipo de algumas variáveis difere do tamanho delas, fazendo com que seja necessário que esses dados sejam especificamente alinhados. Assim, no caso de uso de diversos tipos de dados, é importante observar esse alinhamento.

Como dito anteriormente, para se obter um desempenho maior, é necessário que se reduzam as trocas de informação, uma vez que a troca de dados entre a CPU e o FPGA é constante e cada byte transmitido tem um custo, que pode impactar no desempenho geral. Em conjunto a isso, há também a necessidade de ter conhecimento do espaçamento entre os dados. Geralmente, a cada instrução do computador, lê-se um certo número de bytes em conjunto, a fim de reduzir o tempo gasto nas leituras e escritas, uma vez vez que a leitura de byte a byte é muito mais demorada. Cada tipo de variável possui um tamanho em bytes diferente; assim, um número *int* geralmente utiliza 4 bytes, um *double* utiliza 8 e um *short int* utiliza apenas 2. Assim sendo, o computador tenta alinhar sempre esses dados, colocando espaçamento nos que não cabem no restante do conjunto de dados lido, promovendo um aumento na quantidade de memória (espaço) utilizado. Com isso, é importante organizar os dados de forma que um conjunto, também chamado de *chunk*, de dados lido seja mais eficiente possível.

2.12 CONCLUSÕES

Neste capítulo, foram definidos os elementos fundamentais para a compreensão da pesquisa. Foram analisados, inicialmente, o sistema de potência e os conceitos de carregamento de VE, para que em seguida, fossem apresentados os AG e as ferramentas que seriam utilizadas para solucionar o problema proposto. Por fim, é feita uma análise dos conceitos necessários para se aplicar o paralelismo no modelo proposto.

Por se tratar de um escopo bem específico, não foram encontrados muitos trabalhos com uma relação considerável com esta pesquisa. Existem trabalhos que tratam de coordenação de carregamento de VEs utilizando métodos meta-heurísticos, como os AGs, além de trabalhos que utilizam o FPGA para otimização e técnicas de HIL, entretanto não foram encontrados muitos trabalhos que apresentem esses temas em conjunto.

Capítulo 3

Metodologia

Para resolver o problema proposto, inicialmente, foi definido seu escopo, juntamente com suas considerações e em seguida, foi definido o modelo de AG utilizado. São também apresentadas as funções *fitness* e penalidade dos AG, além das restrições aplicadas ao problema. As restrições têm como base as equações apresentadas por (RUEDA-MEDINA et al., 2013).

3.1 CONSIDERAÇÕES DO PROBLEMA

Para este problema, é considerado que:

- O sistema de distribuição é equilibrado e sua otimização é realizada em seu equivalente monofásico.
- Os VEs são conectados e desconectados ao sistema em horários pré-definidos.
- A função *fitness* engloba o custo do sistema e penalidades em caso de não cumprimento dos requisitos do sistema. A função *fitness* deve ser minimizada e sua saída, ao final da análise deve ser igual ao custo total do sistema.
- O tipo de carregamento dos VEs considerado é a carga lenta, que consiste no fornecimento de uma potência monofásica de até 4 kW.
- O GDs considerado nessa pesquisa são síncronas, obedecendo à curvas de capabilidade do gerador síncrono.

O sistema de distribuição é considerado equilibrado e representado em seu equivalente monofásico, pois o sistema de teste escolhido possui essas mesmas características. Além disso, os GD são considerados síncronos pois apresentaram bons resultados e, outros tipos de geradores podem ser modelados como síncronos. As penalidades da função *fitness* são aplicadas, nos casos que não forem atendidos os seguintes itens:

- Ao final do ciclo de 24h, os VEs deve estar com pelo menos 95% de suas cargas.
- A qualquer hora, que a subestação deve ter um fator de potência superior ao pré-estabelecido.

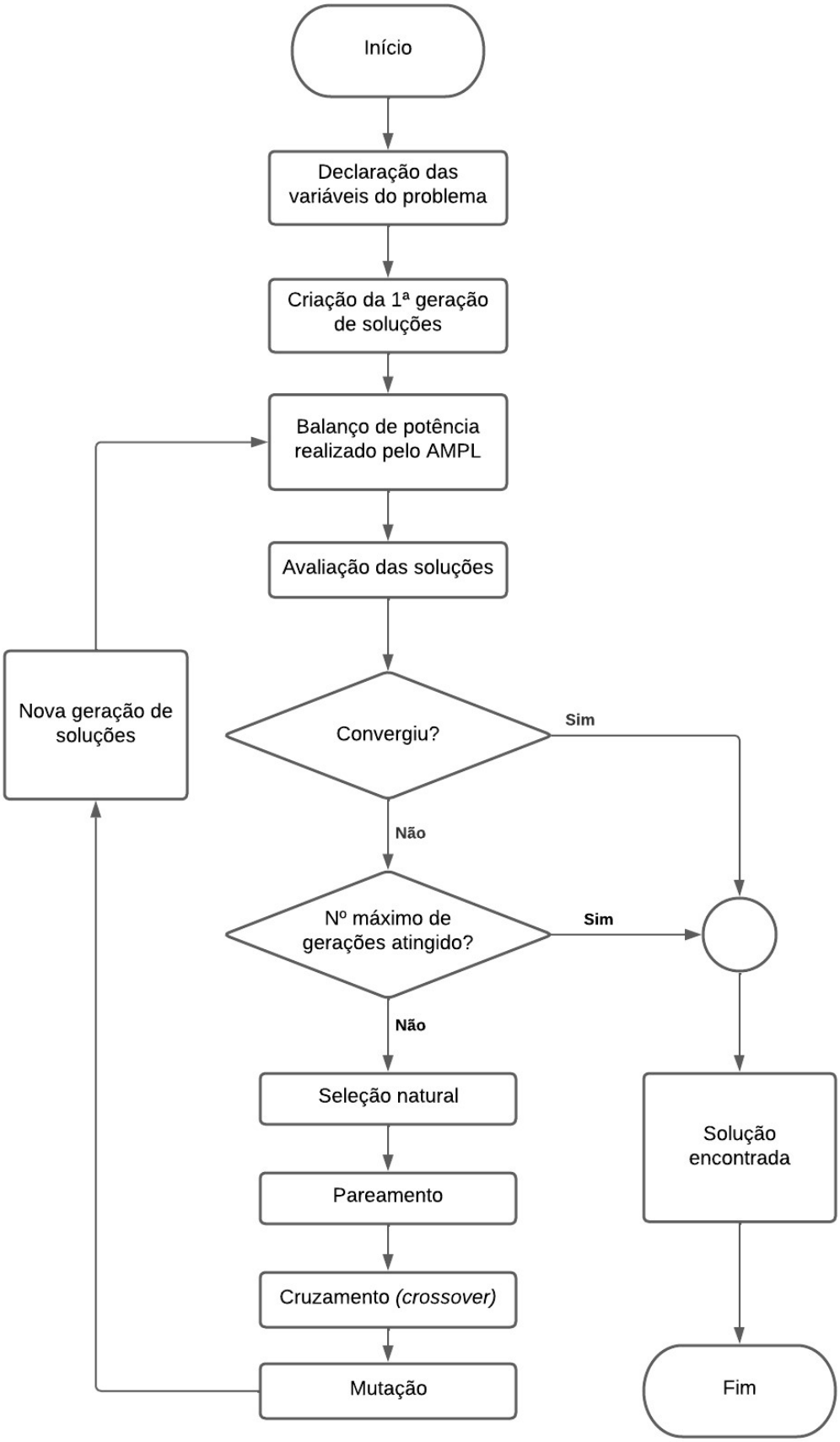
- A qualquer hora, a tensão ou corrente devem estar dentro do limite pré-estabelecido.
- Em todos os períodos de tempo, o balanço de potência deve ser respeitado, garantido pelo fluxo de potência e pelos limites pré-estabelecidos.
- Em cada instante, o fator de potência da subestação deve estar dentro dos limites propostos.

3.2 SOLUÇÃO DO PROBLEMA UTILIZANDO ALGORITMOS GENÉTICOS

Para a implementação do método de AG, foi utilizada a linguagem de programação C++, enquanto que, para resolver o problema de fluxo de potência e garantir o balanço de potência nodal do sistema, foi utilizada a ferramenta AMPL, uma vez que esse balanço possui maior complexidade para ser implementado em C++.

Na Figura 10, é apresentada a estrutura utilizada para implementar o método de AG. As soluções são compostas por um vetor, que contém os valores gerados pelos AG. O vetor é composto, inicialmente, pelas potências de carregamento dos VEs por barra por unidade de tempo $PEV(bus, VE, t)$, seguido das potências ativas e reativas de cada GD, também por unidade de tempo $Pdg(dg, t)$ e $Qdg(dg, t)$.

Figura 10 – Diagrama do processo de otimização do Algoritmo Genético



Fonte: Próprio autor.

Na Tabela 2, é apresentado esse vetor, onde cada linha da tabela corresponde a uma parte do vetor solução.

$PEV(b_1, e_1, t_1)$	$PEV(b_1, e_1, t_2)$	...	$PEV(b_n, e_n, t_n)$
$Pdg(dg_1, t_1)$	$Pdg(dg_1, t_2)$	...	$Pdg(dg_n, t_n)$
$Qdg(dg_1, t_1)$	$Qdg(dg_1, t_2)$	...	$Qdg(dg_n, t_n)$

Tabela 2 – Vetor solução do cromossomo

3.3 FORMULAÇÃO MATEMÁTICA

3.3.1 Função *fitness* do problema de otimização

A função *fitness*, contendo os custos operacionais e as penalidades, é mostrada na Equação 2.

$$\begin{aligned} &\sum_{t=1}^{T}[C_{ss}(t) \cdot P_{ss}(t)] + \sum_{t=1}^{T}\sum_{dg=1}^{DG}[C_{dg} \cdot Pdg(dg,t)] + \\ &Fpen_{ss} \cdot |OL_{ss}| + Fpen_V \cdot \sum_{i=1}^{N_{busses}} |OL_V(i)| + \\ &Fpen_I \cdot \sum_{i=1}^{N_{branches}} OL_I(i) + Fpen_{VE} \cdot \sum_{VE=1}^{N_{VE}} NC(VE) \end{aligned} \tag{2}$$

Onde C_{ss} e C_{dg} são os custos operacionais da subestação e dos GDs respectivamente, o P_{ss} e P_{dg} são as potências ativas da subestação e GDs respectivamente. As penalidades são definidas por $Fpen_{ss}$, $Fpen_V$, $Fpen_i$ e $Fpen_{VE}$, que são respectivamente relativas a subestação, tensão, corrente e VE. As quantidades que estão fora dos limites pré-estabelecidos são representadas por OL_{ss}, OL_V, OL_i e NC, que são respectivamente relacionados a subestação, tensão, corrente e VEs.

Na Equação 2, é possível ver que o custo operacional é dado pelo somatório no tempo do custo da subestação multiplicada pela potência ativa e do somatório no tempo dos custos dos GDs, multiplicada também por sua potência ativa fornecida. O somatório das penalidades é calculado e multiplicado pelo fator de penalidade $Fpen$ correspondente de cada um.

3.3.2 Balanço de potência

De forma a garantir o balanço de potência, foi calculado o fluxo de potência do sistema, fazendo com que as potências ativas e reativa injetadas fossem iguais às potências de saída somadas à demanda do sistema. As Equações 3 e 4 (CESPEDES, 1990)

mostram a representação convencional do balanço de potência, onde a carga do nó é a somatória da demanda e do carregamento dos VEs.

$$\sum_{j=1}^{N_{busses}} P(j,i,t) - \sum_{j=1}^{N_{busses}} [P(i,j,t) + R(i,j) \cdot I^2(i,j,t) + P_{ss}(t) + \sum_{dg=1}^{DG} Pdg(dg,t) = P_d(i) \cdot fd(t) + \sum_{VE=1}^{N_{VE}} \sum_{j=1}^{Nbusses} c(j,VE,t) \cdot PEV(j,VE,t) \quad (3)$$

$$\sum_{j=1}^{N_{busses}} Q(j,i,t) - \sum_{j=1}^{N_{busses}} [Q(i,j,t) + X(i,j) \cdot I^2(i,j,t)] + Q_{ss}(t) + \sum_{dg=1}^{DG} [Qdg(dg,t)] = Q_d(i) \cdot fd(t) \quad (4)$$

Onde P e Q são as potências ativas e reativas respectivamente, R, X I são a resistência, reatância e corrente no ramo, respectivamente. As potências ativa e reativas demandadas, são respectivamente P_d e Q_d, enquanto que o fator de demanda é fd, a variável binária de conexão é dada por c, a potência de carga dos VEs é PEV e as potências reativas da subestação e dos GD são respectivamente Q_{ss} e Q_{dg}.

Em seguida, pelas Equações 5 e 6 (CESPEDES, 1990), é possível obter os níveis de tensão e corrente do último nó, a partir dos valores de tensão do primeiro nó, fluxo de potência ativa, reativa, magnitude da corrente e parâmetros elétricos de cada ramo que conecta os nós.

$$V^2(i,t) - 2[R(i,t)P(i,j,t) + X(i,j)Q(i,j,t)] - Z(i,j) \cdot I^2(i,j,t) - V^2(j,t) = 0 \quad (5)$$

$$I^2(i,j,t) \cdot V^2(j,t) = P^2(i,j,t) + Q^2(i,j,t) \quad (6)$$

Onde V e a Z são a tensão do barramento e a impedância do ramo, respectivamente.

3.3.3 Penalizações

Considerando que os dados dos AG são fornecidos ao AMPL para que esse ele forneça o resultado do fluxo de potência, é necessário aplicar penalidade aos resultados que não correspondem aos limites regulatórios ou especificados. Isso significa que o AMPL é responsável apenas por obter a resposta do fluxo de potência, sem restrições,

enquanto que o AG é responsável por verificar se os limites estão sendo obedecidos a partir dos dados que ele mesmo forneceu. Caso esses limites não sejam obedecidos, o AG gera uma nova população de cromossomos (ou soluções) e reenvia para o AMPL, realizando uma nova análise.

Nas Equações 7 e 8 correspondem à positividade do fornecimento da subestação e do fator de potência a partir das potências ativas e reativas da subestação.

$$Pss(t) \geq 0 \tag{7}$$

$$-P_{ss}(t) \cdot \tan(\cos^{-1}(fp_{min})) \leq Q_{ss}(t) \leq P_{ss}(t) \cdot \tan(\cos^{-1}(fp_{max})) \tag{8}$$

Onde fp_{min} e fp_{max} são respectivamente os fatores de potência mínimo e máximo pré-estabelecido.

Também, são aplicadas penalidades às correntes em cada ramo do sistema e a tensão em cada nó, em caso de descumprimento dos limites regulatórios. São mostrados esses limites nas equações 9 e 10.

$$0 \leq I^2(i,j,t) \leq I^2_{max}(i,j) \tag{9}$$

$$V^2_{min} \leq V^2(i,t) \leq V^2_{max}(i,t) \tag{10}$$

Onde I_{max} é a corrente máximo do ramo e V_{min} e V_{max} são as tensões mínimas e máximas de cada barramento.

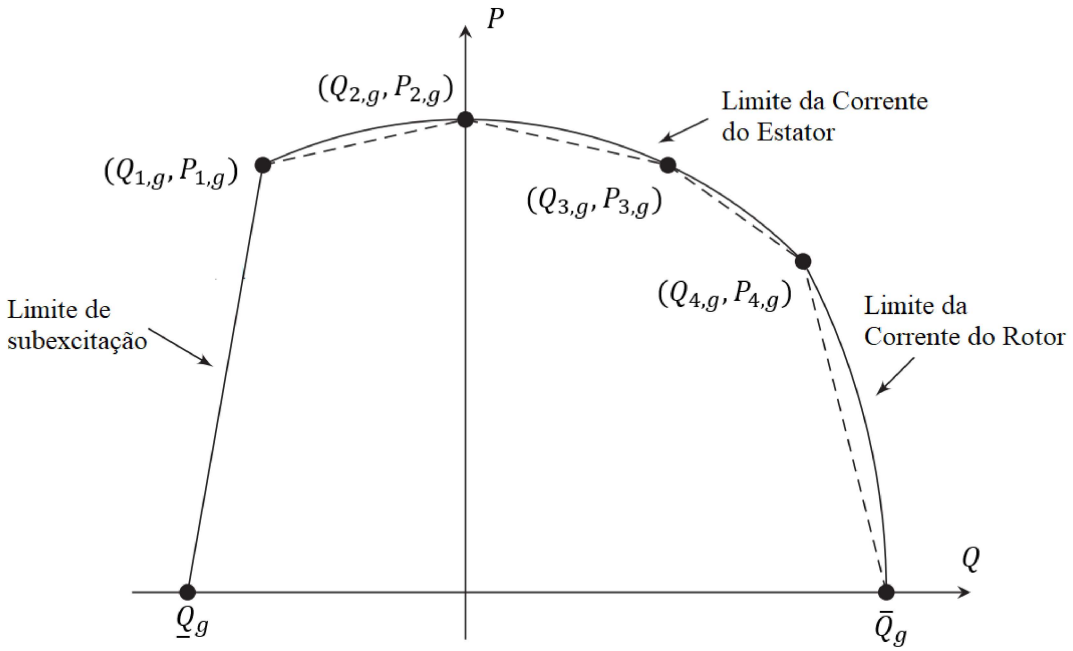
Por fim, foi adicionada uma penalização para as variáveis de carregamento dos VEs, nos AG. Essa penalização ocorre quando os VEs não alcançam o carregamento mínimo previamente definido, fazendo com que os AG procure soluções que possuam o carregamento previsto. Essa penalização é proporcional à quantidade de energia que for menor do que o carregamento mínimo.

3.3.4 Restrições de limite de geração de potência dos geradores síncronos e fator de potência

Para esta pesquisa, é considerado que os GDs são do tipo síncrono. Dessa forma, os GDs podem absorver ou fornecer potência reativa para a rede e devem fornecer potência ativa para o sistema. Para obedecer a esses limites, foi limitada a variação dos valores do vetor solução para que essa permanecesse dentro de sua curva de capacidade de geração. A Figura 11 mostra os limites de geração que devem ser respeitados.

Nota-se que a capacidade de fornecer ou absorver potência reativa é dada pela corrente de campo do gerador. Mesmo possuindo uma característica não linear, é possível delimitar a curva em retas, facilitando o processamento das restrições e mantendo-se um resultado satisfatório.

Figura 11 – Curva de capacidade de geração do gerador



Fonte: (RUEDA-MEDINA et al., 2013)

A restrição apresentada na Equação 11 garante que a potência ativa fornecida pelos GDs seja sempre positiva e dentro do seu limite máximo de fornecimento. As restrições relativas às retas de linearização das curvas de capacidade dos GDs são apresentadas nas Equações 11 à 16 correspondem aos respectivos limites da curva de geração.

$$0 \leq Pdg(dg,t) \leq Pdg_{max}(dg,t) \tag{11}$$

$$Pdg(dg,t) \leq \left(\frac{Pdg_1(dg)}{Qdg_1(dg) - Q(dg)_{min}(dg)}\right) \cdot [Qdg(dg,t) - Qdg_{min}(dg)] \tag{12}$$

$$Pdg(dg,t) \leq \left(\frac{Pdg_2(dg) - Pdg_1(dg)}{Qdg_2(dg) - Q_1(dg)}\right) \cdot [Qdg(dg,t) - Qdg_2(dg)] + Pdg_2(dg) \tag{13}$$

$$Pdg(dg,t) \leq \left(\frac{Pdg_3(dg) - Pdg_2(dg)}{Qdg_3(dg) - Q_2(dg)}\right) \cdot [Qdg(dg,t) - Qdg_3(dg)] + Pdg_3(dg) \tag{14}$$

$$Pdg(dg,t) \leq \left(\frac{Pdg_4(dg) - Pdg_3(dg)}{Qdg_4(dg) - Q_3(dg)}\right) \cdot [Qdg(dg,t) - Qdg_4(dg)] + Pdg_4(dg) \quad (15)$$

$$Pdg(dg,t) \leq \left(\frac{Pdg_4(dg)}{Qdg_4(dg) - Q(dg)_{max}(dg)}\right) \cdot [Qdg(dg,t) - Qdg_{max}(dg)] \quad (16)$$

Onde, os valores de $Pdg_1(dg)$ a $Pdg_4(dg)$ e $Qdg_1(dg)$ a $Qdg_4(dg)$, $Qdg_{min}(dg)$ e $Qdg_{mx}(dg)$, corresponde aos respectivos limites da curva de geração.

3.3.5 Demanda de potência de carregamento dos VEs

O carregamento dos VEs é limitado pela quantidade de energia fornecida pela estação de carregamento e, por isso, é necessária uma restrição que englobe a positividade do fornecimento, além da máxima potência fornecida ao VE. Assim como no caso da limitação da variação das potências nos GDs, o vetor solução dos AG limita esse valor de potência de carga dos VEs. Essas limitações são mostradas nas seguintes equações. A Equação 17 garante que ambas as restrições sejam atendidas.

$$0 \leq PEV(bus, VE, t) \leq PEV_{max} \quad (17)$$

Onde PEV_{max} é a potência máxima de carregamento do VE.

Na equação 18, é apresentada a restrição que limita a variação máxima permitida pela estação de carregamento em qualquer intervalo de tempo. A variação de um intervalo de tempo $t-1$ e t deve ser de no máximo ΔPEV_{max}

$$\Delta PEV \leq \Delta PEV_{max} \quad (18)$$

Onde ΔPEV e ΔPEV_{max} são respectivamente a variação de potência de carregamento do VE e a variação máxima de potência de carregamento do VE.

Por fim, as restrições apresentadas nas equações 19 a 21 são relativas ao estado de carga das baterias dos VEs. As equações mostram respectivamente que o estado inicial deve ser igual à carga da bateria quando o VE é conectado ao sistema; a carga da bateria no estado t deve ser igual a carga do estado $t-1$ somada a potência PEV, que por sua vez, é multiplicada pela eficiência de carregamento; e que o estado de carga da bateria seja menor ou igual a potência nominal da bateria e não nulo.

$$c(j, VE, t_{conn}) \cdot EV(VE, t_{conn}) = c(j, VE, t_{conn}) \cdot EV_{nom}(VE) \cdot \frac{S_i(VE)}{100} \quad (19)$$

$$c(j,VE,t)\cdot EV(VE,t) = c(j,VE,t)\cdot EV(VE,t-1)+c(j,VE,t)\cdot PEV(j,VE,t)\cdot\eta \quad (20)$$

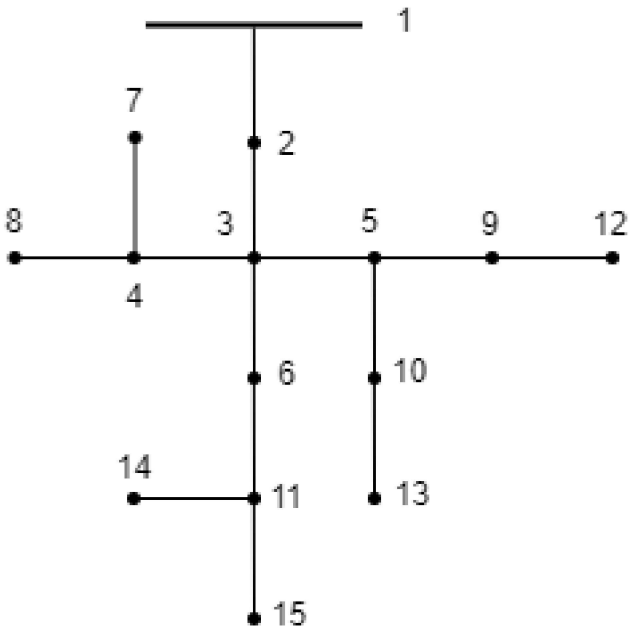
$$0 \leq c(j,VE,t)\cdot EV(VE,t) \leq c(j,VE,t)\cdot EV_{nom}(VE) \quad (21)$$

Onde EV e EV_{nom} são a carga do VE e a carga nominal do VE, respectivamente. A variável S_i diz respeito a percentagem de carga inicial do VE.

3.4 SISTEMA DE TESTES UTILIZADO

Foram realizados testes e simulações de forma a validar o método de coordenação e otimização do carregamento dos VEs. As simulações foram realizadas utilizando um sistema de distribuição radial de média tensão (13,8 kV), com 15 barras, adaptado do sistema IEEE 37 nós, cujos parâmetros são mostrados na Tabela 3. O nó de número 1 é considerado a referência do sistema, logo a tensão deste possui valor nominal e um ângulo de zero graus. O sistema é representado na Figura 12.

Figura 12 – Sistema IEEE 37 nós adaptado



Fonte: Próprio autor.

B_i	B_j	R(Ω)	X(Ω)	I_{max}	P_j	Q_j
1	2	1,19392	1,22615	800	210	105
2	3	0,86346	0,89846	800	0	0
3	4	1,80447	0,58011	800	0	0
3	6	0,83753	0,47759	800	0	0
3	5	1,18726	1,23539	800	28,33	13,33
5	9	1,08265	0,34806	800	81	44.67
5	10	1,39586	0,79598	800	78,33	43,33
11	14	0,36084	0,116	800	144	74
11	15	1,86114	1,0613	800	28,33	13,33
4	7	1,44364	0,46411	800	0	0
4	8	1,08265	0,34806	800	14	7
6	11	1,20977	0,68986	800	0	0
9	12	0,65137	0,37144	800	12,67	6
10	13	0,46529	0,26533	800	128,33	63,33

Tabela 3 – Parâmetros do sistema

Nessa rede, foram alocados, de forma aleatória, 2 GDs nos nós de números 3 e 10, sendo que esses possuem capacidade de potência ativa máxima de 250 kW e potência reativa entre -250 kvar a 188,75 kvar e -147 kvar e 222,625 kvar, respectivamente.

Em relação aos VEs, foram selecionados, de forma aleatória, 7 nós, para alocar carregadores de 5 VEs em cada um deles, totalizando 35 VEs no sistema. A potência nominal de cada VE é considerada 20 kW, que é um valor típico para VEs, enquanto a taxa de carregamento (ΔPEV) e a sua variação máxima de carregamento PEV_{max} são iguais a 4 kW e 1 kW, respectivamente. Os intervalos de tempos considerados são de uma hora em um período de 24 horas. O estado inicial das baterias dos VEs, quando conectados, foram escolhidos de forma aleatória, entre 5% e 20% e foi definido que os VEs devem estar carregados em, no mínimo, 95% do seu valor nominal até o momento de sua desconexão.

3.5 APLICANDO O PARALELISMO

De forma a compreender melhor o funcionamento do *software* Vitis, foram propostas sete simulações para entender a aplicabilidade dos métodos de paralelismo, utilizando um algoritmo simples de somatória entre vetores. Primeiramente, foi apresentada uma simulação de um sistema sem otimizações, utilizando o FPGA como responsável por ler os dados enviados da CPU, fazer a operação de somatório entre cada um dos elementos do vetor e, por fim, enviar os dados para a CPU.

Em seguida, foram feitas operações simples, como a execução de um problema simples: a soma de dois vetores no FPGA, além de instanciar novos núcleos lógicos centrais (*kernels*), que são responsáveis por permitir a interação entre o *software* e o *hardware*. Foram comparadas todas as simulações, discutindo o impacto no código de cada uma das técnicas aplicadas. Com base nos resultados obtidos, foi proposto um AG simples e que pudesse representar a melhoria que o paralelismo proporciona para esse tipo de problema.

Por fim, foi proposto um modelo de AG com uma função em paralelo para utilizar no FPGA, utilizando o conhecimento adquirido através do estudo e das simulações, aplicando os conceitos para atingir um resultado melhor do que se utilizasse o algoritmo na CPU, sem paralelismo da função em questão. É necessário ressaltar que, neste trabalho, o método de AG não foi implementado no FPGA para solucionar o problema inicialmente proposto, entretanto, foi possível exemplificar o potencial da aceleração de *hardware*, utilizando o FPGA.

3.6 CONCLUSÕES

Neste capítulo, foram realizadas as considerações do problema e suas limitações. Em seguida, apresentou-se a formulação matemática necessária para implementar o algoritmo, além de diagramas do sistema e dos dados do mesmo. Por fim, foram definidas simulações para representar o uso do algoritmo e como atingir os resultados esperados.

Capítulo 4

Resultados e discussões

A partir da metodologia apresentada, foram ensaiados 4 casos da rede elétrica apresentada, de forma a comparar a eficiência da técnica proposta. Os casos são:

- Caso 1: sem GDs ou VEs;
- Caso 2: sem GDs, mas com VEs;
- Caso 3: com GDs, mas sem VEs;
- Caso 4: com GDs e VEs.

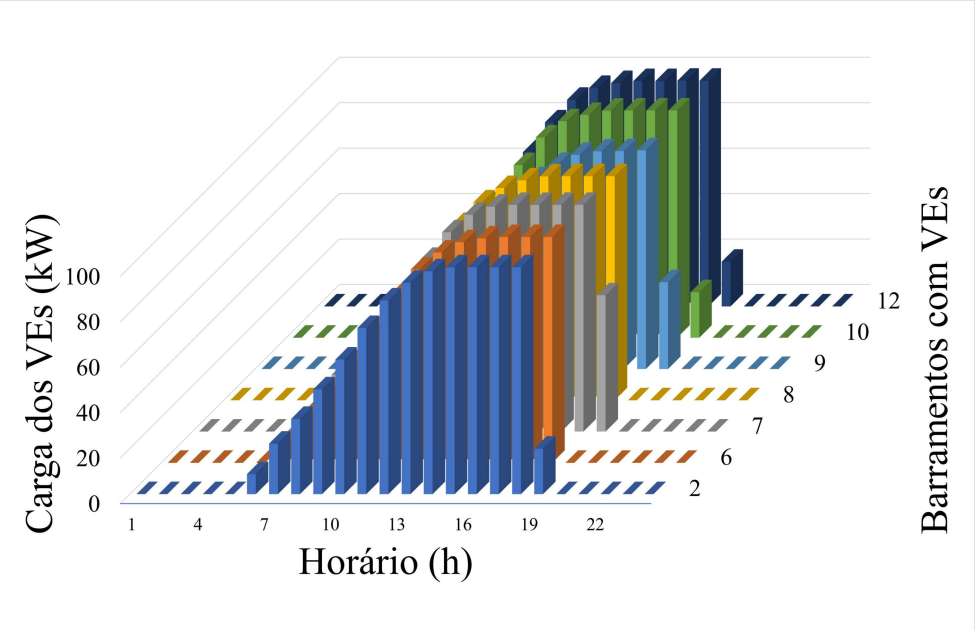
Os valores de potência de carregamento de cada VE, as potências ativas e reativas dos GDs foram obtidas através dos AG, enquanto, o resultado do fluxo de potência foi obtido pelo AMPL, fornecendo os valores de potências ativa e reativa da subestação, além da tensão de cada nó e corrente nos ramos, a cada hora do dia. O Caso 1 foi considerado o sistema referência em relação aos demais. Os resultados de cada caso são apresentados na Tabela 4.

Casos	Custo (R$)	Custo relativo	Perdas (kWh)
Caso 1	204.442	—	107,56
Caso 2	213.237	4,30%	116,71
Caso 3	164.742	-19,42%	35,74
Caso 4	179.512	-12,19%	39,05

Tabela 4 – Custo operacional e perdas

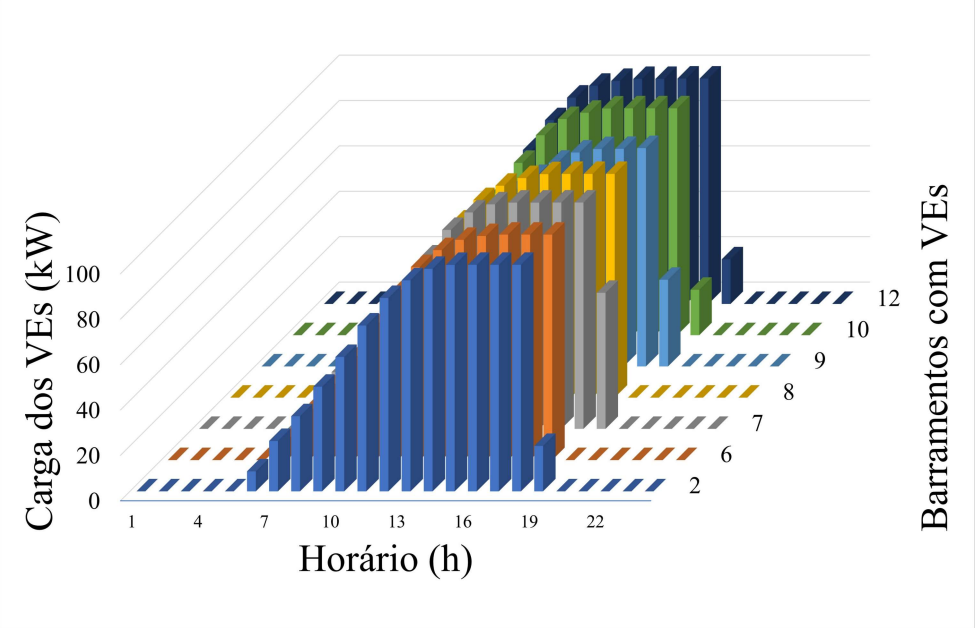
O custo operacional do sistema completo otimizado foi de R$ 179.512, reduzindo o custo do sistema 12,19% em relação ao sistema original.

Figura 13 – Carregamento dos VEs no Caso 2



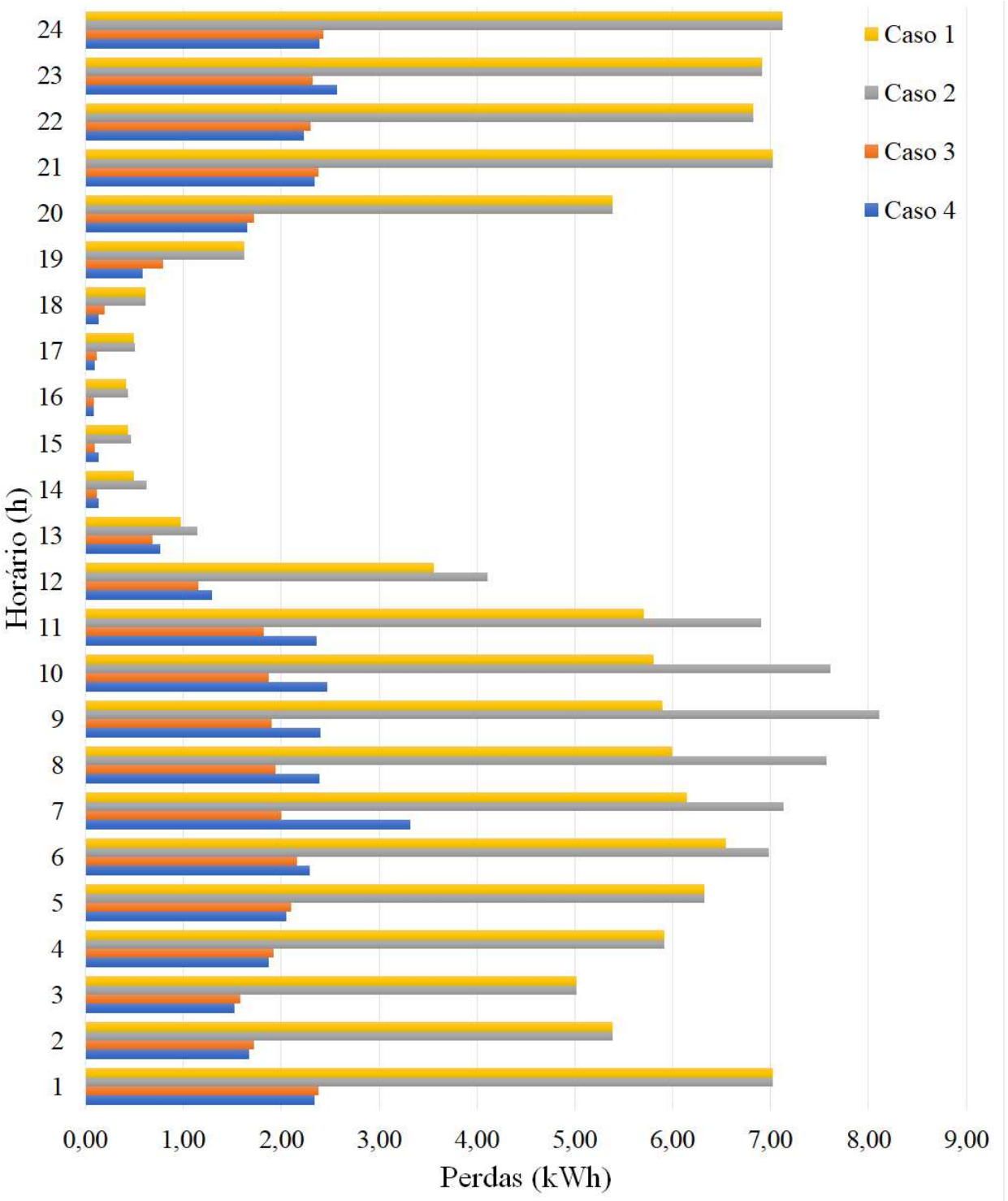
Fonte: Próprio autor.

Figura 14 – Carregamento dos VEs no Caso 4



Fonte: Próprio autor.

Figura 15 – Perdas do sistema a cada hora



Fonte: Próprio autor.

Observando as Figuras 13 e 14, é possível notar que todos os veículos obtiveram um carregamento maior ou igual ao proposto e, ao final, alguns já haviam se desconectado da rede. Na Figura 15, são mostrados os valores de perdas hora a hora nos 4 casos. Nota-se que ao incluir os GDs, há uma queda significativa nas perdas da rede, mesmo com a inserção dos VEs.

Nas Tabelas 5 e 6, são mostrados os valores de potência ativa e reativa e seus custos de geração, respectivamente, da subestação Pss e dos GDs, em cada período. Em ambos os casos apresentados nas Tabelas 5 e 6, nota-se que mesmo com os GDs, que possuem custo menor que as subestações, ainda foi necessário utilizar energia provinda do alimentador. Isso ocorre, pois para que fosse respeitado o balanço de potência, era necessário que a subestações também fornecesse potência reativa e, consequentemente, potência ativa, para que também fosse respeitada as restrições de fator de potência. Também é contabilizado a potência ativa e reativa dos GDs, porém esses são limitados pela sua curva de capacidade de geração.

Na Tabela 5 durante os períodos de 14 horas às 18h e na Tabela 6 às 14, 17 e 19 horas, nota-se que o algoritmo levou a solução que fornece potência ativa apenas do GD 2, uma vez que essa era suficiente para abastecer o sistema, já que nesses períodos de tempo, as baterias dos VEs já se encontravam em um estado de carga relativamente alto, e reduzindo seus consumos, como também é possível observar nas Figuras 13 a 15.

t	P_{ss}	$P_{ss\$}(R\$)$	P_{dg1}	$P_{dg1\$}(R\$)$	P_{dg2}	$P_{dg2R\$}(\$)$
1	304,90	15,88	176,25	12,89	234,63	10,47
2	221,55	16,1	175,00	12,55	231,55	10,45
3	200,73	15,57	175,00	12,73	230,50	10,66
4	249,35	16,03	175,00	12,44	232,96	10,41
5	270,91	15,71	175,00	12,86	234,04	10,82
6	281,35	15,14	175,00	12,7	234,57	10,72
7	260,48	15,23	175,00	12,56	233,52	10,61
8	253,52	15,75	175,00	12,63	233,17	10,55
9	247,96	16,54	175,00	12,65	232,89	10,51
10	243,09	15,63	175,00	12,64	232,64	10,36
11	237,53	16,35	175,00	12,96	232,36	10,63
12	110,63	15,73	175,00	12,44	225,90	10,34
13	0,00	15,43	28,11	12,57	239,38	10,34
14	0,00	16,33	0,00	12,44	193,69	10,69
15	0,00	16,89	0,00	12,55	180,62	10,66
16	0,00	16,45	0,00	12,54	175,54	10,32
17	0,00	15,42	0,00	12,47	195,15	10,74
18	0,00	16,33	0,00	12,89	213,33	10,8
19	0,00	16,53	110,04	12,99	235,83	10,38
20	221,55	15,81	175,00	12,64	231,55	10,83
21	304,90	16,14	176,25	12,51	234,63	10,5
22	296,59	15,52	175,79	12,78	234,63	10,69
23	299,36	15,08	175,94	12,52	234,63	10,77
24	309,06	16,51	176,48	12,57	234,63	10,79

Tabela 5 – Geração de potência ativa – Caso 3

t	P_{ss}	$P_{ss\$}(R\$)$	P_{dg1}	$P_{dg1\$}(R\$)$	P_{dg2}	$P_{dg2\$}(R\$)$
1	330,82	15,88	209,79	12,89	175,12	10,47
2	242,32	16,1	214,93	12,55	170,81	10,45
3	208,17	15,57	206,45	12,73	191,54	10,66
4	266,99	16,03	210,98	12,44	179,29	10,41
5	302,3	15,71	203,22	12,86	174,4	10,82
6	328,81	15,14	219,79	12,7	169,42	10,72
7	451,87	15,23	71,01	12,56	207,28	10,61
8	356,54	15,75	210,05	12,63	167,82	10,55
9	352,78	16,54	218,35	12,65	171,62	10,51
10	371,72	15,63	209,21	12,64	169,86	10,36
11	352,4	16,35	211,18	12,96	177,6	10,63
12	198,95	15,73	210,1	12,44	181,68	10,34
13	17,79	15,43	219,17	12,57	79,44	10,34
14	42,03	16,33	0,00	12,44	179,33	10,69
15	20,13	16,89	77,57	12,55	96,11	10,66
16	3,4	16,45	38,97	12,54	135,54	10,32
17	25,61	15,42	0,00	12,47	169,8	10,74
18	3,35	16,33	30,56	12,89	179,72	10,8
19	176,31	16,53	0,00	12,99	169,61	10,38
20	232,00	15,81	225,04	12,64	171,01	10,83
21	321,63	16,14	211,04	12,51	183,07	10,5
22	303,32	15,52	214,84	12,78	188,77	10,69
23	342,23	15,08	210,35	12,52	157,59	10,77
24	339,52	16,51	209,44	12,57	171,16	10,79

Tabela 6 – Geração de potência ativa – Caso 4

4.1 APLICANDO SIMULAÇÃO EM TEMPO REAL

Após realizar obter os resultados apresentados na seção anterior, foi avaliado seus tempos de simulação. Por ser um AG, a execução da simulação dispendeu de um tempo elevado, sendo necessário aproximadamente 20 horas para obter um resultado otimizado. Esse tempo extrapola o passo de tempo proposto nesse trabalho, que era realizado de hora em hora, o que leva a necessidade de melhorar o desempenho do algoritmo. Assim, foi proposto a aplicação da simulação em tempo real utilizando o FPGA pois a aceleração de *hardware* reduz o tempo de execução, como é mostrado no 2, deixando a simulação mais rápida.

A seguir, são apresentados a simulações e testes realizados no FPGA para aplicar o paralelismo de execução e assim, alcançar a simulação em tempo real.

4.2 FPGA E VIVADO/VITIS

O primeiro passo realizado para obter a aceleração do algoritmo, foi criar um *kernel* para aplicar ao FPGA, utilizando a ferramenta Vivado em sua nova versão, mostrada na Figura 4. Seguindo os passos propostos na pesquisa (MEDINA, 2019), foi possível configurar esse núcleo para rodar os algoritmos realizados em C/C++. Entretanto, de primeira instância, não foi possível rodar algoritmos mais complexos.

Após pesquisar o motivo de não ser possível realizar operações mais complexas, foi descoberto que, atualmente, o Vivado faz parte de um pacote de *softwares* oferecidos pela Xilinx, chamado Vitis, e que foi seccionado para atender a escopos de soluções distintas. Como mencionado previamente no Capítulo 2, o Vivado é uma ferramenta para programação de *hardware*, enquanto que o Vitis atende ao desenvolvimento para *software*. Apesar de diferentes, os *softwares* podem ou não trabalhar de forma complementar, a depender a necessidade do programador.

A primeira versão do Vitis foi lançada no em janeiro de 2020 e os materiais para sua utilização eram escassos. Além disso, mesmo após algum tempo de lançamento, o método de instalação ainda possui uma complexidade relativamente elevada, em comparação a *softwares* tradicionais, apenas sendo mais intuitivo a partir da versão lançada em 2022.

Como previamente descrito no Capítulo 2, mesmo com a ferramenta Vitis, que possibilita a programação em linguagens com uma abstração maior que as que são desenvolvidas para *hardware* (Verilog e VHDL), ainda é necessário adaptar o código e seus paradigmas. Inicialmente, foi proposto que o programa desenvolvido pudesse ser transcrito para linguagens de máquina, apenas com a utilização da ferramenta, sem mudanças significativas no código. Entretanto, quando estudado de forma mais aprofundada, verificou-se que, além da complexidade relativamente elevada de instalação e utilização do Vitis, ainda era necessário remodelar e reestruturar o código feito previamente.

Após realizada a instalação, que foi feita no sistema operacional (SO) Ubuntu-Linux, foi realizado um teste para se familiarizar com o ambiente do Vitis e verificar o funcionamento do FPGA. Esse SO foi escolhido para que fosse possível a comunicação entre os núcleos lógicos no FPGA e da CPU. Caso outro SO tivesse sido escolhido, algumas funcionalidade (como instanciar um *kernel* do Linux) não seriam possíveis (XILINX, 2022a).

A seguir, foram realizadas simulações utilizando o FPGA. Inicialmente, utilizou-se o ZedBoard Zynq-7000 para se somar 2 vetores, entretanto, por falta de suporte do fabricante, foi optado por utilizar o simulador de *hardware* do Vitis. Nessa etapa, foram simuladas a adição entre dois vetores sem otimização e, gradualmente, foram

apresentadas simulações utilizando as otimizações previamente descritas no Capítulo 2. Ao final, foram aplicadas as otimizações no AG em uma função, mostrando o potencial do uso de um FPGA.

4.2.1 Adição de dois vetores no ZedBoard Zynq-7000

O primeiro teste realizado para verificar o desempenho do FPGA foi a adição entre dois vetores de tamanho 1024, quatro vezes seguidas. Nesse caso, foi utilizado diretamente o FPGA, considerando um *clock* de 100 MHz e comparado com uma CPU tradicional de 3,7 GHz, sendo uma velocidade de 37 vezes maior que o FPGA. O código utilizado para essa experiência é apresentado no Apêndice, na Seção A.1.

Entretanto, nesse primeiro momento, os resultados não foram de acordo com o esperado. O tempo de execução realizado pela CPU foi de 19 μs, enquanto que no FPGA foi de 149,37 μs. Apesar do resultado ser menor, notou-se que não é proporcional à diferença de *clock*, uma vez que a CPU possui 37 vezes mais ciclos de *clock*, porém, forneceu um resultado aproximado de apenas 8,7 vezes mais rápido.

De forma a obter um resultado melhor e mais rápido, foram colocadas as diretrizes *pragma* contendo o conjunto de otimizações previamente estabelecidas pelo compilador GCC (GCC, 2022). Os *pragma* são diretrizes utilizadas para instruir o compilador a maneira que ele irá processar o código fornecido, sendo possível extrair vantagens da capacidade do compilador, otimizando o código. Essas otimizações não são feitas por padrão e devem ser especificadas. Isso é necessário pois o compilador procura fornecer compilações mais rápidas e depuráveis (*Debug*), tornando o programa menos ágil. As diretrizes fornecem informações ao compilador para que seja abdicado da possibilidade de depurar o código e que o tempo de compilação seja mais elevado. Realizando esse processo, obteve-se um resultado melhor, sendo possível executar o programa em aproximadamente 29,15 μs.

Em seguida, foram pesquisadas novas formas de otimização para utilizar no FPGA em questão (ZedBoard Zynq-7000), porém, não foram encontrados materiais suficientes. Isso acontece pois o *software* Vitis, atualmente na versão 2.5, não fornece mais suporte, exemplos ou novas imagens de núcleos lógicos para a família Zynq-7000, desde a versão 1.4. A justificativa para isso é que o Vitis tem o foco em inteligencia artificial e o *hardware* em questão, apesar de seu uso ser tecnicamente possível, seu desempenho para esse tipo de aplicação é insatisfatório (XILINX, 2021).

4.2.2 Adição de dois vetores utilizando simulação

De forma a validar a melhora de desempenho do paralelismo do programa no FPGA, foi optado pela simulação de *hardware*, ferramenta também oferecida pelo pacote

do Vitis. Com essa ferramenta, é possível verificar o tempo de execução que seria obtido, caso se utilizasse o código no FPGA, além de outras grandezas de interesse. Também é possível obter os processos que ocorrem nos núcleos lógicos do FPGA de forma gráfica, o que permite que o programador simule quantas vezes for necessário até atingir sua meta de otimização, sem precisar gerar o arquivo executável para o FPGA, que possui um tempo muito elevado de compilação.

A simulação consistiu em realizar parte do código na CPU e uma parte específica no FPGA. O FPGA é um *hardware* que é flexível para ser programado, entretanto, não possui um relógio próprio e gerar um número pseudo-aleatório não é eficaz. Além disso, sua memória de armazenamento é reduzida (GARCIA et al., 2019), então, é recomendado que o FPGA seja apenas um servo (*kernel*) e a CPU seja a responsável pela coordenação e armazenamento das variáveis, sendo o mestre (*host*). Isso significa que as operações mais complexas ou que não são facilmente paralelizáveis, são realizadas pela CPU, enquanto que as tarefas paralelas e simples, são feitas pelo FPGA e enviadas para o *host*. Por ser uma simulação de um modelo genérico do FPGA, foi utilizado um *clock* de 150 MHz.

A primeira simulação realizada foi feita para que não se utilizasse otimização, apenas com a soma entre dois vetores de tamanho 1024, quatro vezes (mesmo exemplo da Seção 4.2.1). O código para essa simulação possui grandes diferenças em relação ao anterior, sendo necessário configurar diversos protocolos de comunicação para que seja possível utilizar o FPGA em conjunto com a CPU.

O primeiro passo foi a verificação da lista de *hardwares* disponíveis na lista de *devices*. Assim, foi preciso criar um contexto, que é responsável por todas as operações entre o *device* e o *host* e em seguida carregar explicitamente o arquivo executável do *kernel* no código e instanciá-lo no FPGA simulado.

Dentro do contexto, existe um espaço de comunicação entre o *host* e o *device* que é chamado de memória global que serve para troca de dados entre ambos. Após a configuração da conexão entre os dispositivos, foi alocado uma espaço nessa memória para os três *buffers* de comunicação, dois para os vetores de entrada e um para a saída. O *buffer* é um espaço alocado na memória para transferência de dados entre dois dispositivos que possuem taxa de transferências distintas que, nesse caso, é a memória global, utilizada para comunicação entre o FPGA e a CPU.

Por fim, é realizada uma fila de atividades para o *device* e com os dados necessários para realizar as operações. Esse processo de definição de todo contexto, permite que a comunicação entre o *host* e o *device* seja mais eficiente, diminuindo tempos de leitura e escrita diretamente um com o outros.

A seguir, são apresentadas as 7 simulações para somar quatro pares de vetores,

utilizando o FPGA. Na primeira simulação, não há nenhum tipo de otimização, apenas é utilizada a CPU para geração e armazenamento dos vetores e o simulador de *hardware* para executar a adição dentro do FPGA. Em seguida, é aplicada a divisão dos quatro pares de vetor em unidades de processamento distintas, aumentando a velocidade de processamento. Na simulação 3, foi testado se o aumento das unidades de processamento é proporcional ao número delas. Na simulação 4, utilizou-se novamente uma unidade de processamento, porém foi aplicado a técnica de *streaming*, para em seguida, na simulação 5, se utilizar 4 unidades de processamento.

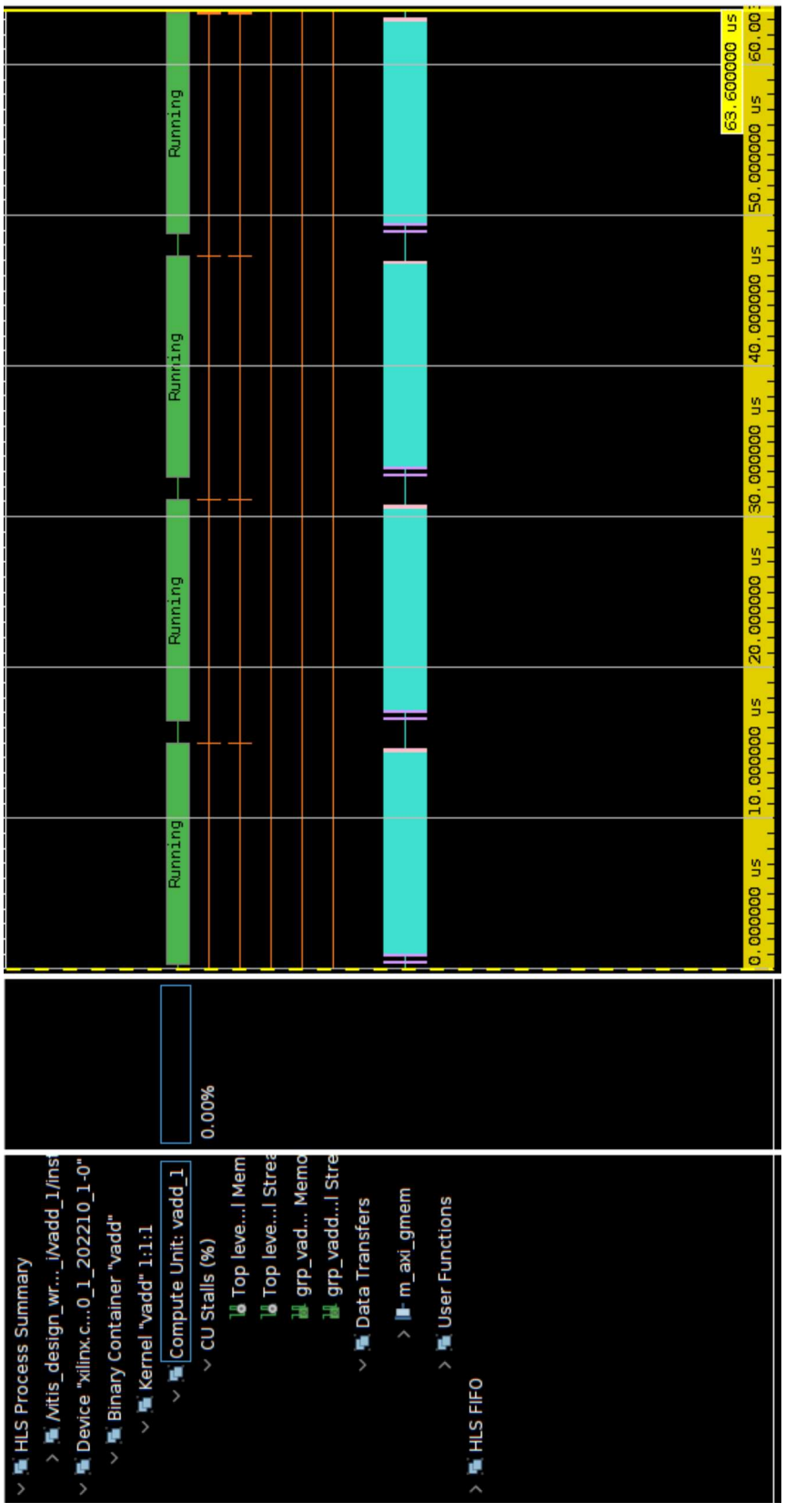
Nas simulações 6 e 7, são então apresentados os comparativos do com e sem a técnica de *streaming*, porém com tamanhos diferentes de dados: na simulação 6 utiliza-se o mesmo vetor de 1024 elementos, porém essa operação é realizada mil vezes seguidas; enquanto que na simulação 7, é feita a comparação com vetores 16 vezes maior que os anteriores, porém executado apenas uma única vez.

4.2.2.1 Simulação 1: sem otimização

Nessa simulação, foi obtido um tempo menor na CPU, de apenas 19 μs, enquanto que o resultado obtido no FPGA simulado foi de aproximadamente 63,6 μs. Foi possível observar que os resultados são piores do que o da CPU, assim como no primeiro experimento utilizando o ZedBoard, o que já se era esperado. Em seguida, não foi utilizada a mesma diretriz (*#pragma GCC optimize*, apresentado no Apêndice A), pois o compilador utilizado para os FPGA que o Vitis ainda fornece suporte, é o v++ para o FPGA e o GCC para compilar o código do *host*, não sendo possível utilizar essa diretriz específica para os próximos códigos que são colocados no FPGA.

Na Figura 16, é possível observar esses resultado. No lado esquerdo, é possível ver os *kernels*, que nesse caso é apenas um instanciado, e os detalhes deles. É possível minimizar ou aumentar o detalhe de cada um dos analíticos do *kernel*. A primeira linha do gráfico, em verde, é o tempo que o *kernel* estava sendo executado e, por isso, na figura são apresentados quatro blocos sequenciais, uma vez que é executada a soma de dois vetores quatro vezes consecutivas. Nas linhas detalhadas abaixo do gráfico verde, é possível observar o tempo de parada (*stall*) do sistema, para escrita na memória. Por fim, no último gráfico, majoritariamente da cor azul, são referentes ao acesso a memória global, sendo a cor azul relativa a leitura e escrita de dados simultaneamente do *kernel*, a roxa é relativa a leitura apenas e a rosa é relativa a escrita do FPGA na memória global.

Figura 16 – Simulação sem Otimização



Fonte: Próprio autor.

4.2.2.2 Simulação 2: com 4 *Computing Units*

A *Computing Unit* (CU) é a instancia de um *kernel* no FPGA. Uma das vantagens do Vitis é que é possível aumentar de forma relativamente fácil o número de instâncias de um mesmo *kernel*, podendo dividir o processamento entre elas. A escolha inicial dessa otimização se dá pela sua simplicidade, com poucas alterações no código e não requer um conhecimento mais profundo sobre as técnicas de otimização em paralelo. As alterações consistem em aumentar o número de *buffers* e do *kernels* instanciados, nas configurações de conexão entre o *host* e *device*.

A Figura 17 mostra o resultado da simulação, sendo como esperado, diminuindo em aproximadamente quatro vezes o tempo de execução em relação a simulação com apenas um *kernel*, resultando em um tempo de conclusão de aproximadamente 15 μs. Dessa forma, foi possível obter um resultado com um tempo um pouco menor do que a CPU, usando a ferramenta Vitis e fazendo apenas com uma otimização relativamente simples. Nessa figura, é possível observar com maior clareza os momentos de escrita e leitura do FPGA e o tempo de parada (stall).

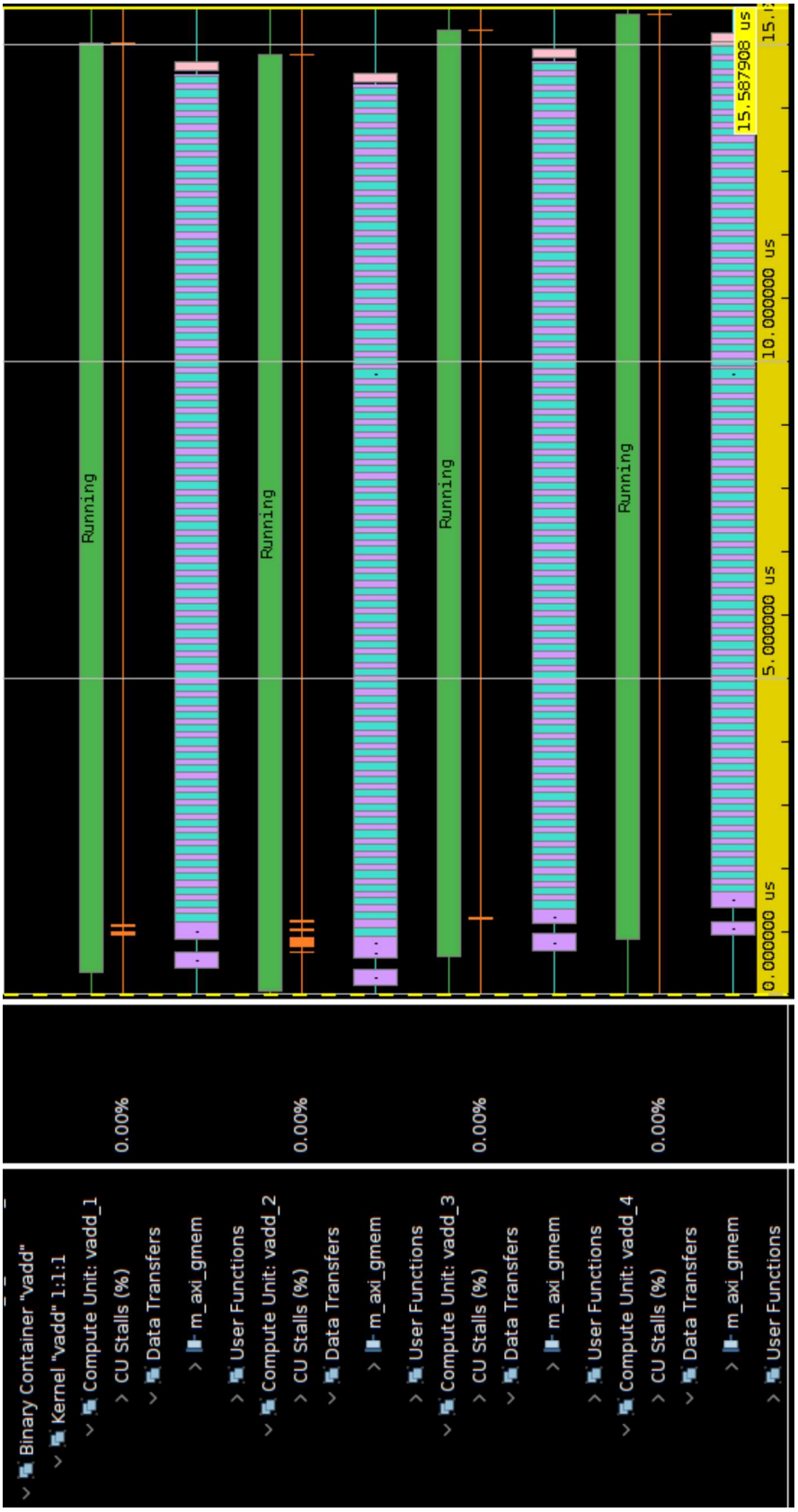


Figura 17 – Simulação com 4 CUs

Fonte: Próprio autor.

4.2.2.3 Simulação 3: com 16 CUs

Após verificar que aumentando o número de CUs resulta em reduções proporcionais no tempo de execução, foi proposto verificar se esse tempo permanece constante, independente do número de *kernels* instanciados. Dessa forma, foi proposta uma simulação utilizando 16 CUs para somar ambos os vetores de tamanho 1024, 16 vezes.

Após a simulação, verificou-se que o resultado foi o esperado, não mantendo-se o mesmo tempo do que quatro CUs. Os resultados são apresentados nas Figuras 18 e 19. Notou-se, então, que o tempo de processamento das 16 operações de soma do vetor, resultou em um tempo maior que a simulação com apenas quatro, resultando em um tempo de aproximadamente 31,1 μs, enquanto que, na CPU, foi de 69 μs. Mesmo que o número de dados computados tenha sido quatro vezes maior e o tempo apenas duas vezes maior aproximadamente, houve uma perda de tempo de processamento, em que as CUs ficaram ociosas. Isso pode ser verificado com maior detalhe na Figura 19, que apresenta os momentos em que houve paradas prolongadas de processamento (*stall*), pois o *hardware* não podia computar simultaneamente tantas operações ou acessar a memória global enquanto o valor não estava pronto, pois estava sendo utilizado por outra CU.

Na Figura 16, é possível observar que não houve momentos de parada consecutivas, pois apenas uma CU estava em execução, enquanto que, na Figura 17, houve apenas breves momentos que as paradas ocorreram, resultando na computação mais rápida da soma dos vetores de 1024 elementos. Isso mostra que nem sempre é interessante aumentar o número de CUs sem critérios pré-definidos, sendo necessário um estudo mais aprofundado do algoritmo e da formas utilizadas para otimizá-lo de forma a reduzir ao máximo o tempo que as CUs ficam ociosas.

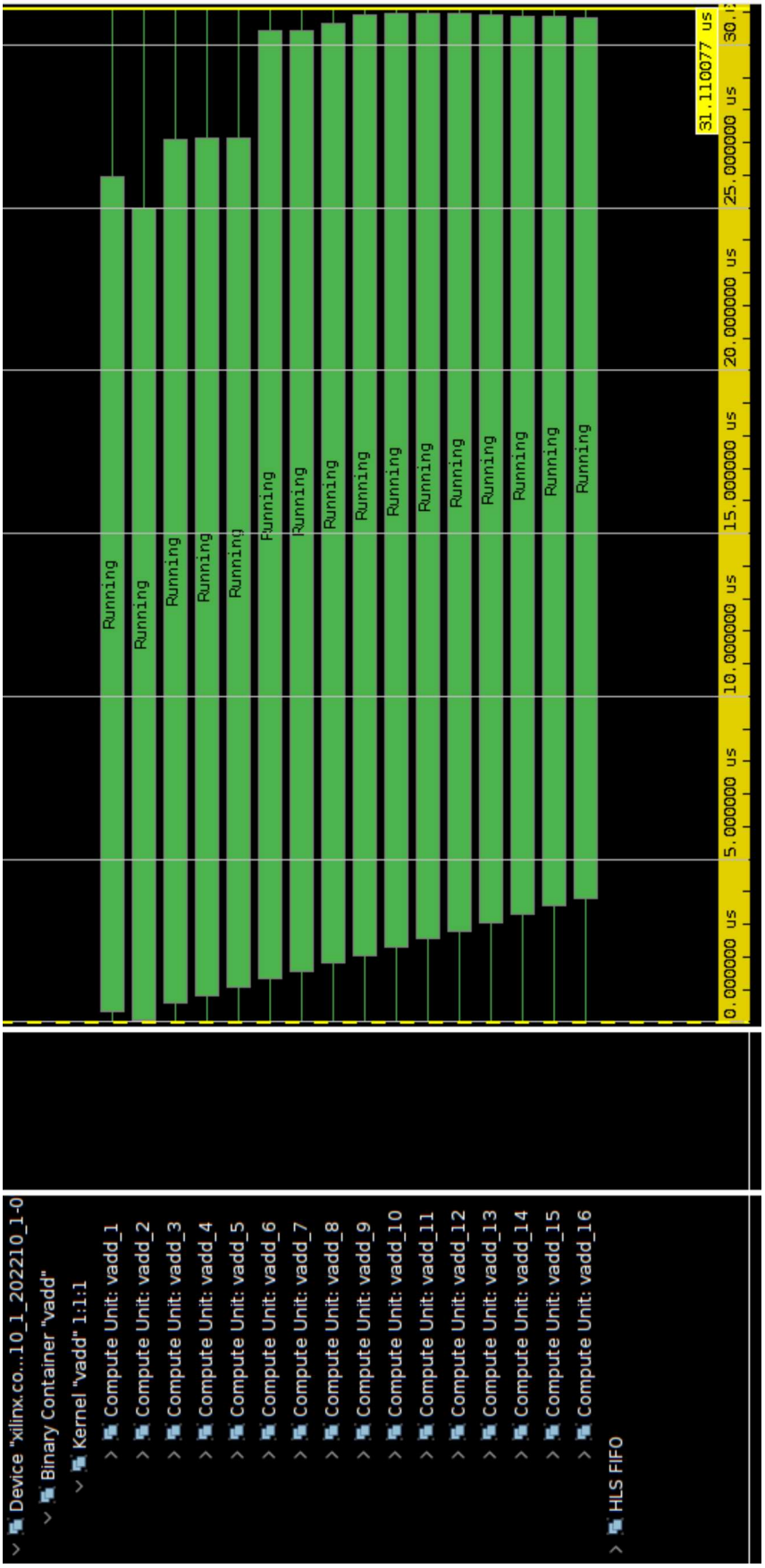


Figura 18 – Simulação com 16 CUs

Fonte: Próprio autor.

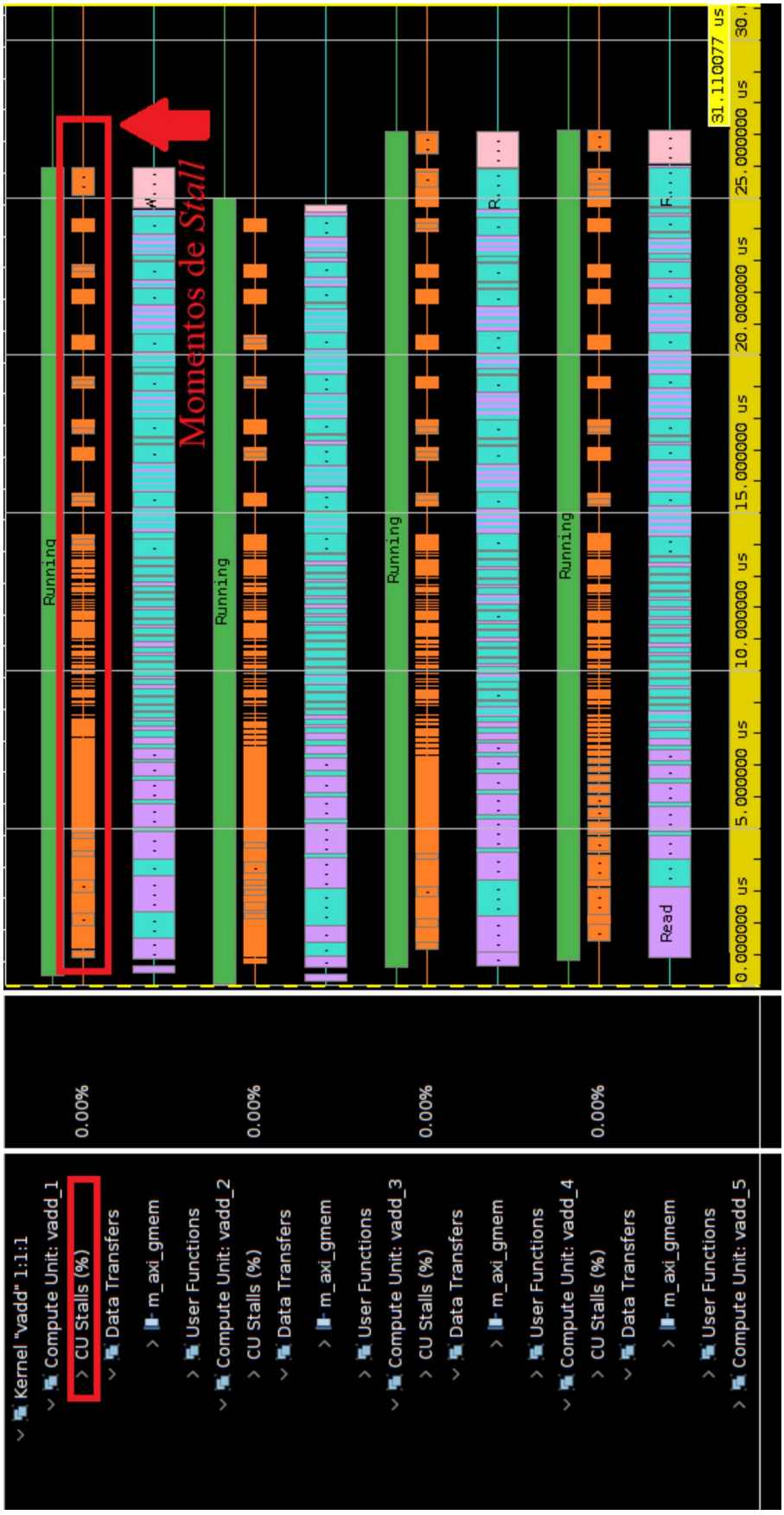


Figura 19 – Detalhe dos momentos de parada da CU (*Stall*)

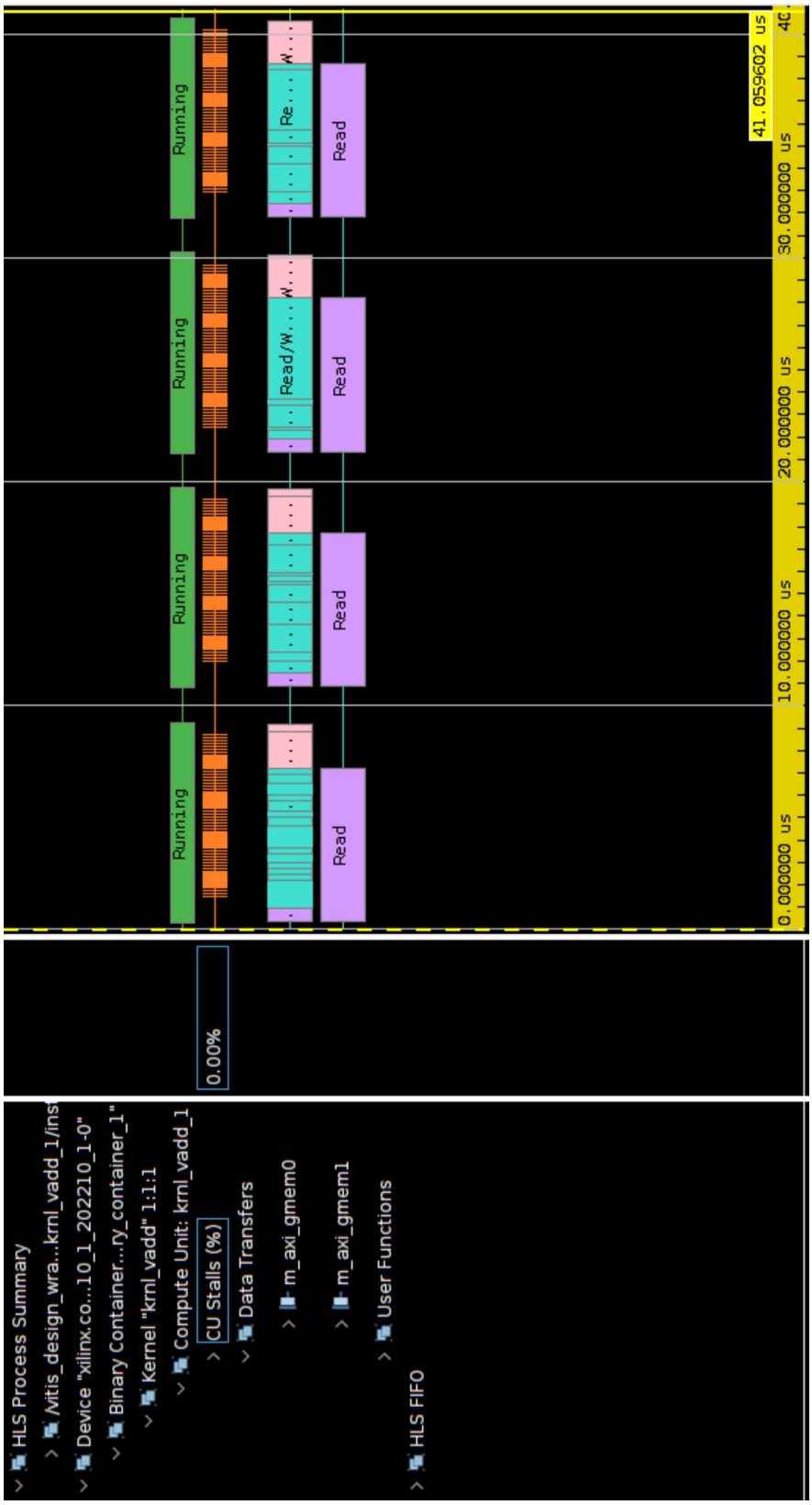
Fonte: Próprio autor.

4.2.2.4 Simulação 4: com 1 CU utilizando diretriz *streaming*

Aplicando os paradigmas de programação em paralelo, apresentados no Capítulo 2, foi necessário alterar o código feito previamente. Geralmente, essa técnica é utilizada para quantidade de dados grandes e contínuas, sendo geralmente o método mais eficiente (XILINX, 2022a). Dessa forma, foram feitas as alterações necessárias no código para que fosse possível utilizar essa técnica. O resultado do *streaming* com uma CU, de um vetor com 1024 é mostrado na Figura 20. O tempo que seria necessário para executar esse código foi de aproximadamente 41 μs, menor do que a simulação com um CU sem *streaming*.

Nesse caso, é possível observar que houve diversos tempo de parada, entretanto o tempo foi relativamente menor do que a simulação 1. Isso ocorre pois, os tempos de parada são muito pequenos e são realizados para que o *host* possa ler o resultado da memória global. Isso significa que ao mesmo tempo que o FPGA processa novos dados de entrada, o *host* já coleta os resultados previamente processados e que estão gravados na memória global. A penúltima linha do gráfico é relativa a memória global que o FPGA lê o valor do vetor 1 e escreve o resultado do cálculo entre o vetor 1 e 2. A última linha é relativa a memória global apenas de leitura do vetor 2, por isso não há nenhum momento de escrita nessa memória global. Dessa forma, é possível já observar ganhos na velocidade de processamento.

Figura 20 – Simulação com 1 CU, utilizando *streaming*



Fonte: Próprio autor.

4.2.2.5 Simulação 5: com 4 CU utilizando diretriz *streaming*

Nessa simulação, foi utilizada a mesma técnica que a anterior, porém, instanciando quatro *kernels*, de forma a reduzir mais ainda o tempo de execução. O resultado da simulação é apresentado na Figura 21. Nota-se que a resposta foi de fato mais rápida que a anterior, porém, não proporcionalmente, sendo necessário cerca de 17,2 μs para executar, sendo mais lento que a simulação com quatro CUs, porém, sem *streaming*. Observando-se os tempos de parada, é notável que em primeiro momento eles são maiores e depois se estabilizam. Dessa forma, é interessante verificar se o tamanho dos dados influencia no tempo de execução total.

Como é possível observar, os tempos de parada são maiores em alguns momentos, fazendo com que esse processamento seja atrasado e o resultado final seja adquirido com maior tempo de execução. Novamente, a primeira memória global é utilizada como leitura e escrita pelo FPGA, enquanto que a segunda é utilizada apenas para leitura.

Figura 21 – Simulação com 4 CU, utilizando *streaming*

Fonte: Próprio autor.

4.2.2.6 Simulação 6: comparação com e sem *streaming*, com 4 CUs

Para realizar essa simulação, foi proposto simular consecutivamente mil vezes ambos os códigos ver qual dos dois permite entregar o resultado em menor tempo, utilizando o vetor de 1024 elementos. O resultado adquirido é apresentado nas Figuras 22 e 23.

Nota-se que o tempo apresentado nas duas simulações são semelhantes, cerca de 19 ms. Isso ocorre, pois existe um tempo até a *stream* estabilizar a conexão e continuar o processo de transferência de dados. Nesse caso simulado, após a transferência dos dados, a conexão é fechada e é necessário ser aberta na *streaming* da iteração seguinte, não sendo aproveitado o benefício de se utilizar a técnica de *streaming*.

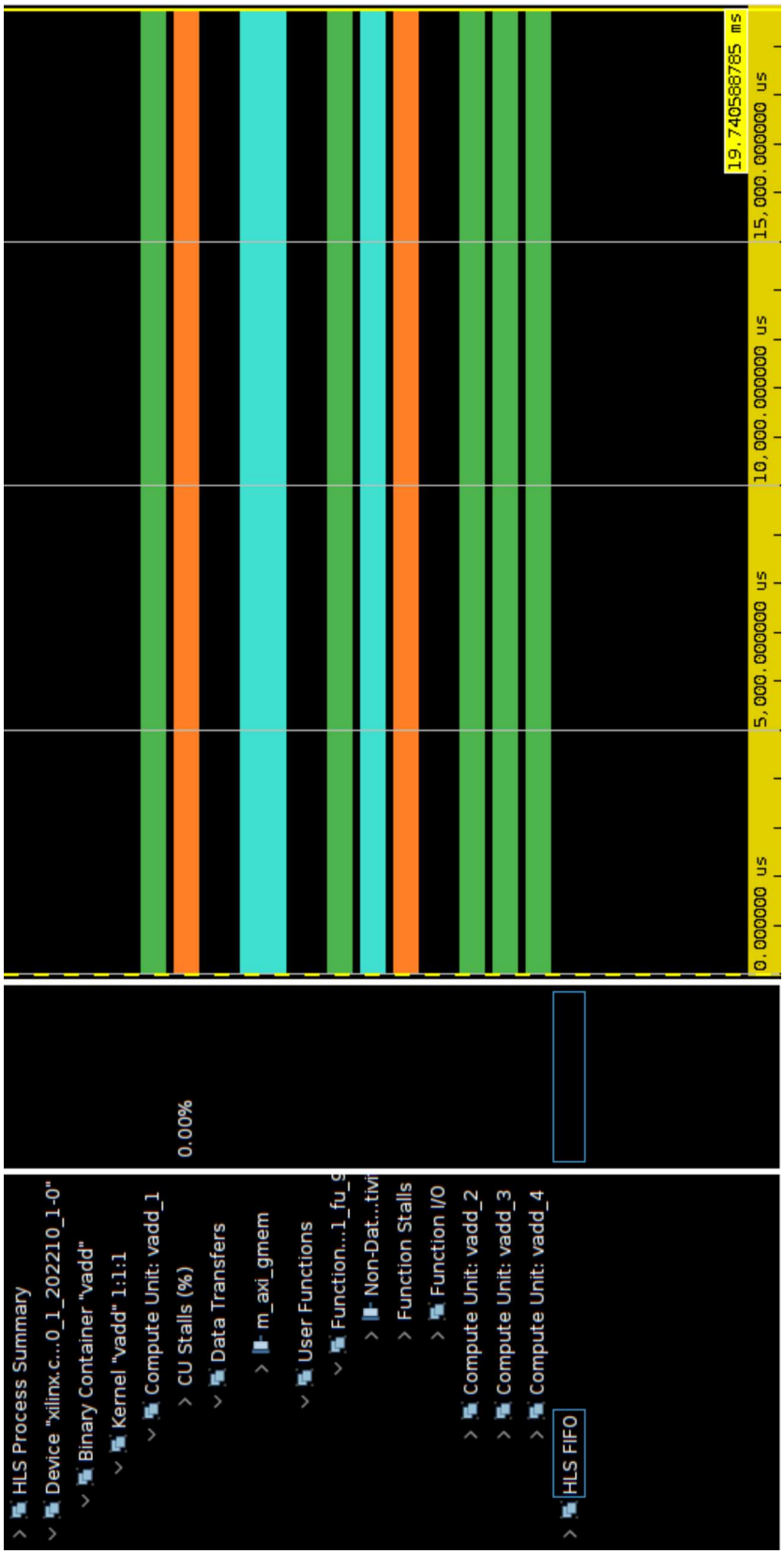
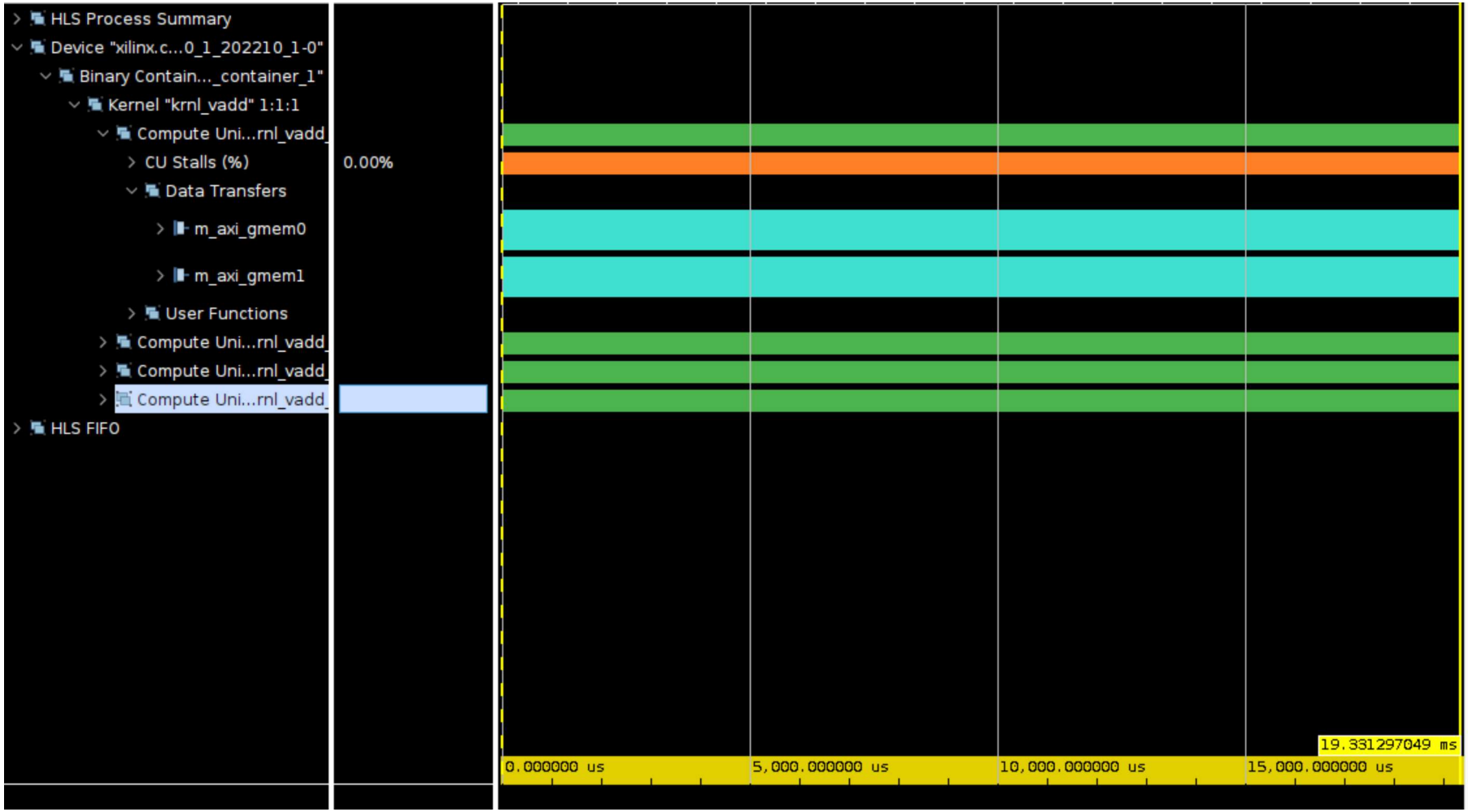


Figura 22 – Simulação com 4 CU sem *streaming* (1000x)

Fonte: Próprio autor.

Figura 23 – Simulação com 4 CU *streaming* (1000x)

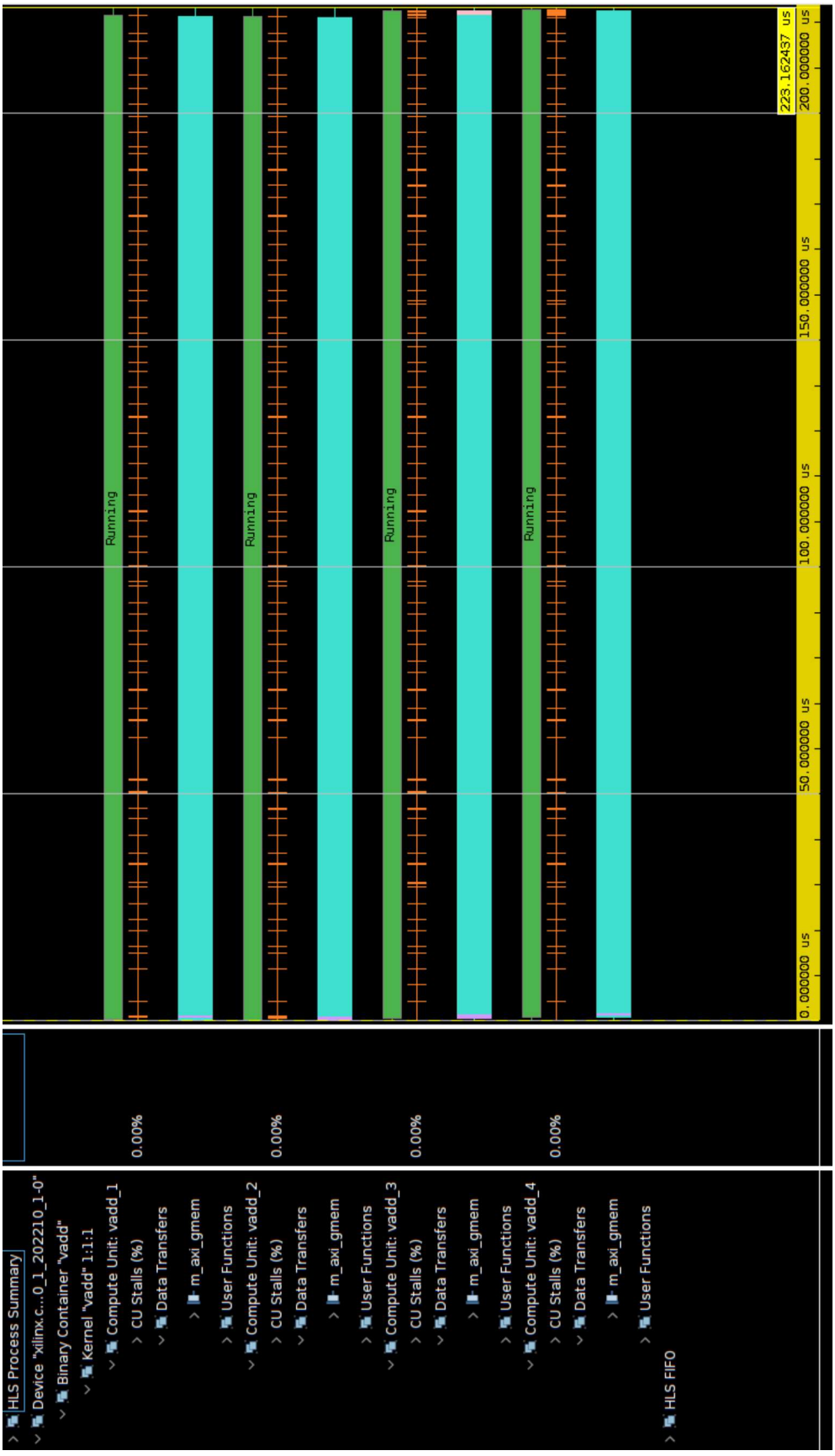


Fonte: Próprio autor.

4.2.2.7 Simulação 7: comparação com e sem *streaming*, com 4 CUs

Outro teste de comparação foi feito com o tamanho dos dados. Nesse caso, foi introduzido um vetor maior, de 16384 elementos, para serem processados tanto no usando o método *streaming* quanto no método sem o *streaming*. O resultado é mostrado nas Figuras 24 e 25.

Analisando os resultado das Figuras 24 e 25, nota-se que o *streaming* teve menor tempo, seguindo a proporção das simulações 1 e 4, sendo mais vantajoso utilizar o streaming, nesse caso. Observando com mais detalhes as imagens anteriores, é possível ver que durante o inicio da *stream* de dados, existem muitas paradas, porém, após algum período é estabilizada a comunicação entre o *host* e os *kernels*, mostrando que o *streaming* é mais rápida nesse caso. A Figura 26 mostra com detalhes essas diferença inicial e depois de estabilizado.



Fonte: Próprio autor.

Figura 24 – Simulação com 4 CU sem *streaming* - 16384 elementos

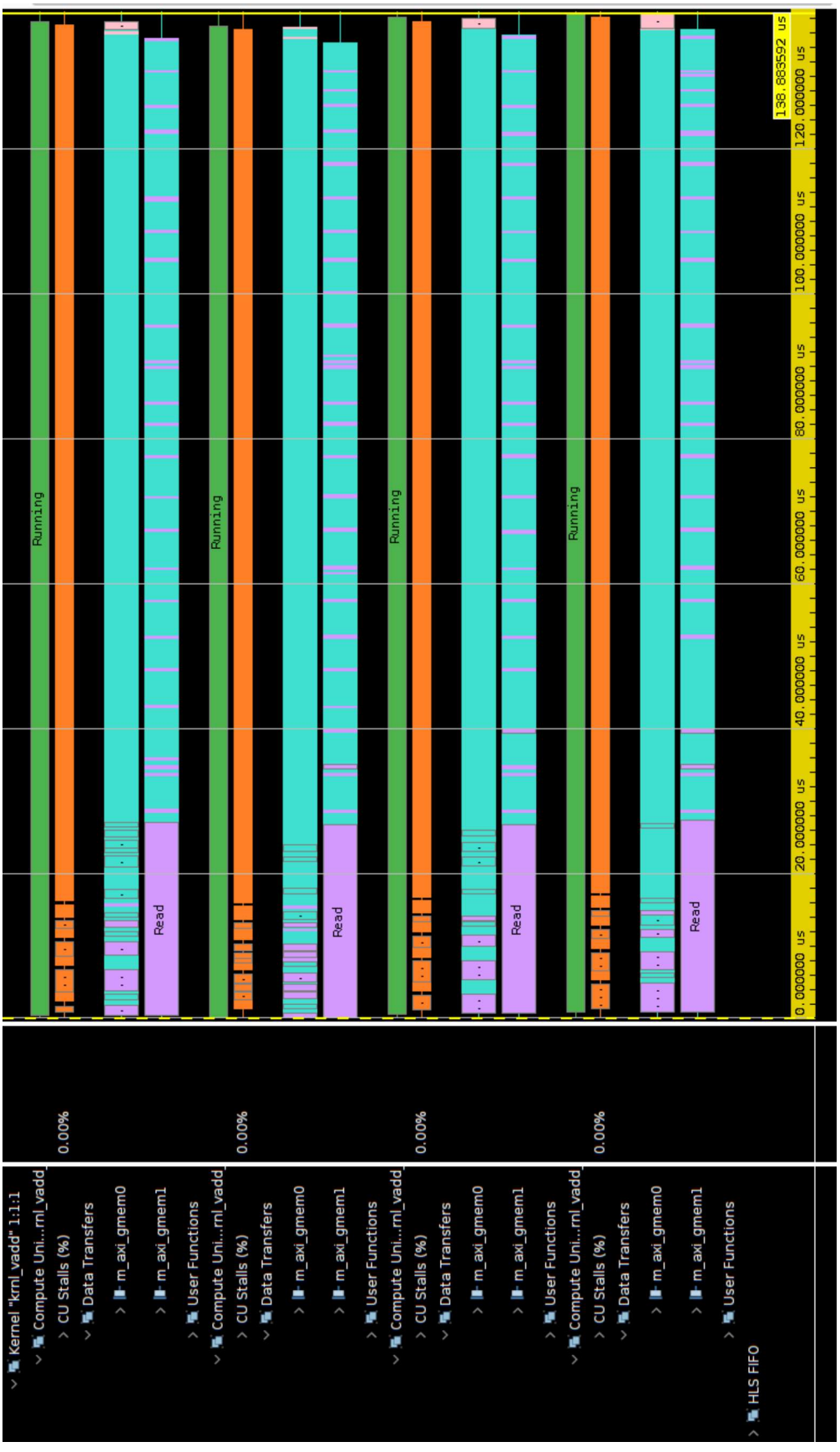


Figura 25 – Simulação com 4 CU *streaming* - 16384 elementos

Fonte: Próprio autor.

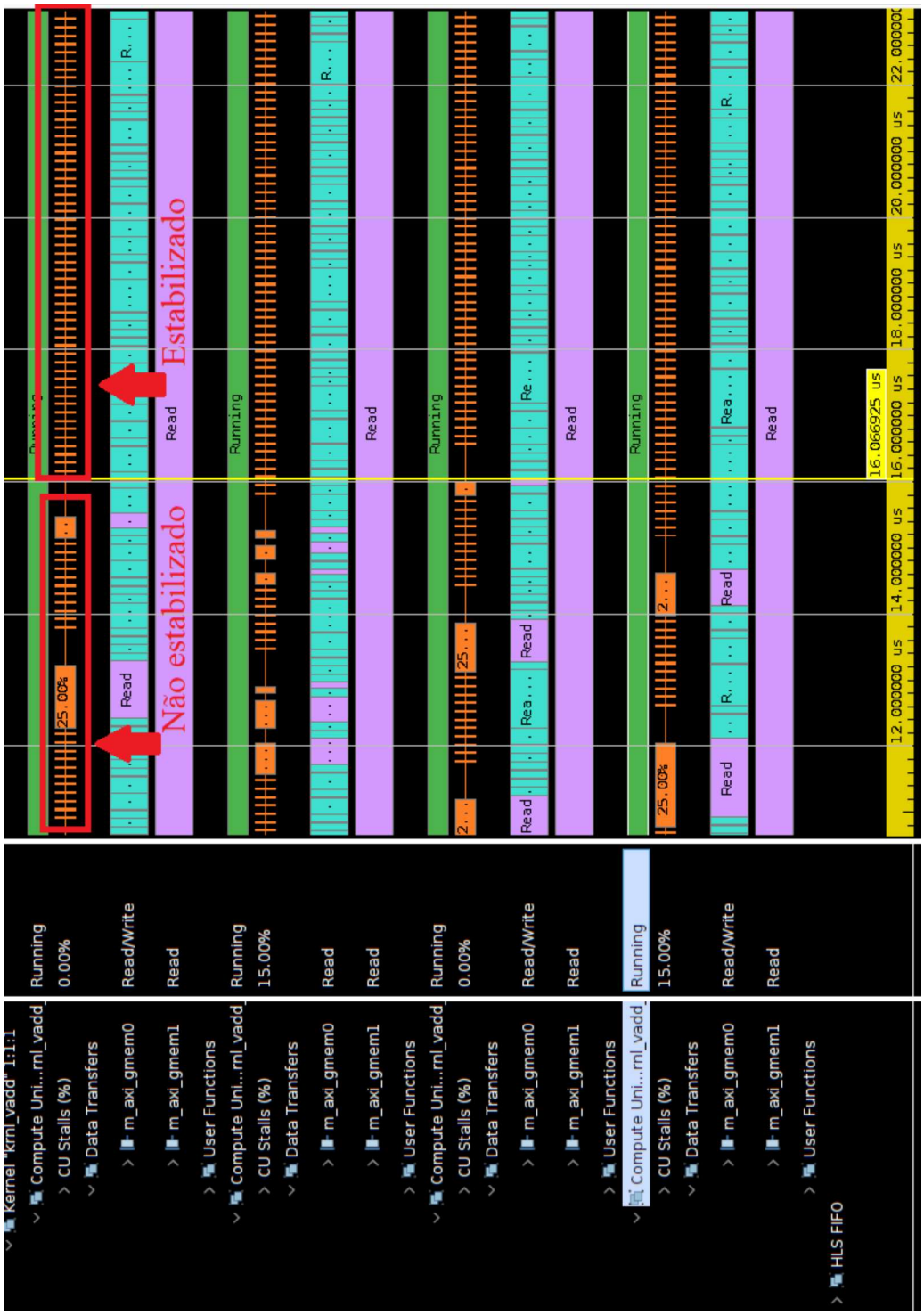


Figura 26 – Simulação com 4 CU *streaming* - Detalhe

Fonte: Próprio autor.

4.2.3 Comparativo entre simulações

Após realizadas as simulações, foi feito um comparativo entre cada uma das otimizações propostas utilizando o FPGA, levando em consideração o tempo que a CPU despende para executar o código. A Tabela 8 mostra o resumo dos tempos de execução das CPU, das simulações utilizando o FPGA e o ganho de redução de tempo de execução em relação aos tempos da CPU.

	Tempo de execução (μs)		Ganho
	CPU	FPGA	
Simulação 1	19	63,6	-234,74%
Simulação 2	19	15	21,05%
Simulação 3	69	31,1	54,93%
Simulação 4	19	41	-115,79%
Simulação 5	19	17,2	9,47%
Simulação 6 s/ *stream*	18052	19741	-9,36%
Simulação 6 c/ *stream*	18052	19331	-7,09%
Simulação 7 s/ *stream*	267	223,2	16,40%
Simulação 7 c/ *stream*	267	138,9	47,98%

Tabela 7 – Comparativo das simulações

Observa-se que o FPGA é mais lento que a CPU, em cerca de 2,34 vezes, para um algoritmo de soma de vetores e sem qualquer otimização. A medida que são aplicadas as técnicas de otimização por aceleração de *hardware*, nota-se um considerável ganho no tempo de execução da função em questão. Os maiores ganhos são mostrados nas simulações 3 e 5, onde possuem uma reduções de 54,93% e 47,98%, respectivamente, o que levou a ambas serem candidatas para a sua aplicação na confecção do AG. As demais simulações foram importantes para ter o embasamento necessário para a escolha e para demonstrar os ganhos que se pode ter ao utilizar do paralelismo e da técnica de HIL.

4.3 MODELAGEM DO AG

Com base nas simulações e nos estudos realizados foram analisadas algumas possibilidades de modelar um AG simples para ser processado paralelamente. O AG em questão, é responsável apenas por gerar a população, avaliar seus cromossomos, realizar o cruzamento e descartar os cromossomos com menor avaliação da geração anterior.

A operação escolhida para ser paralelizada foi a de avaliação dos cromossomos da população, pois essa se mostra facilmente paralelizável, uma vez que cada cromossomo não se correlaciona com os demais na etapa de avaliação.

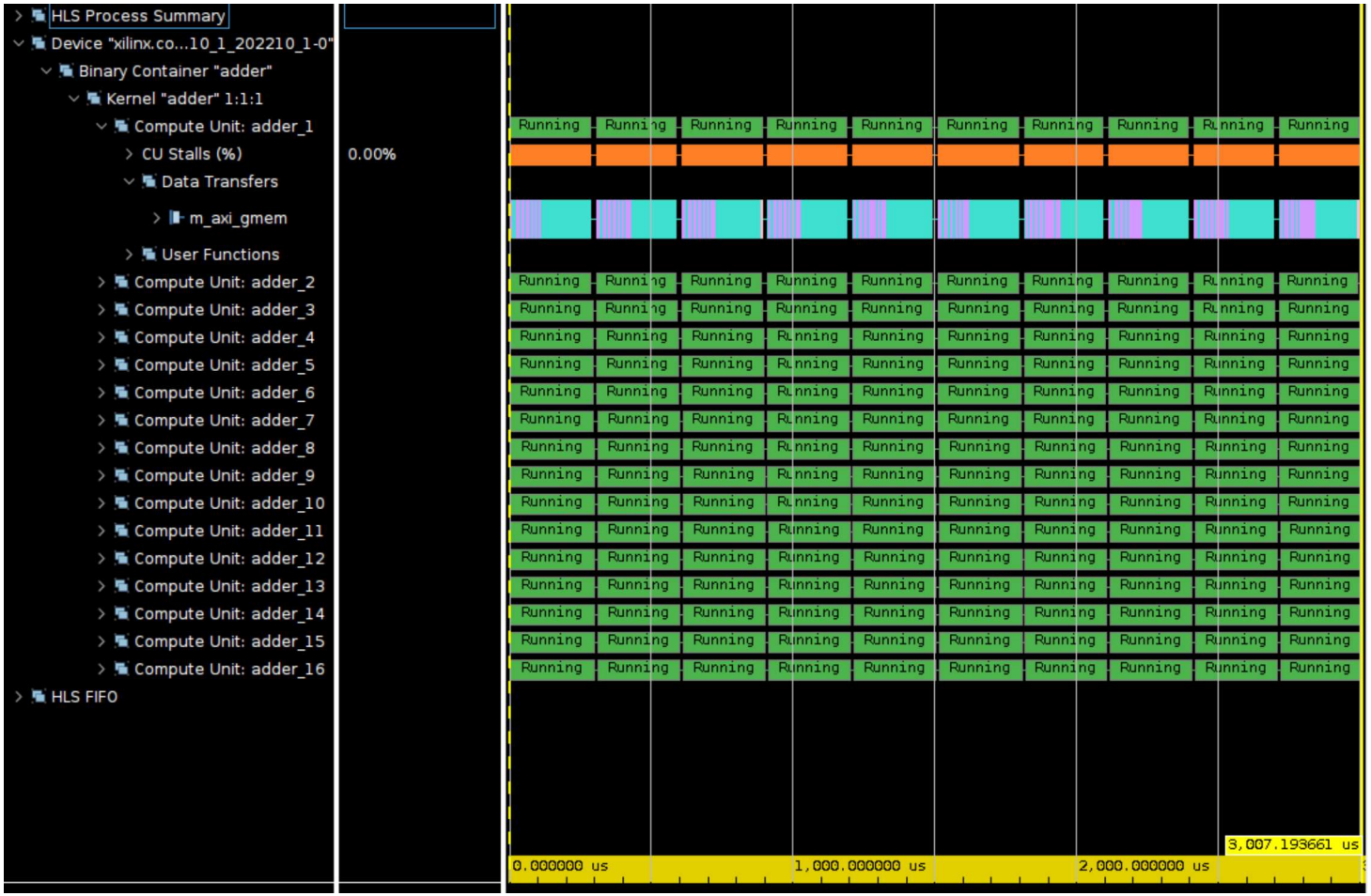
Considerando 10 gerações, uma população de 16 cromossomos, em que cada um possui 16384 elementos, foram aplicadas as técnicas que apresentaram melhor resultado nas simulações: *streaming* e instanciar múltiplos *kernels*. Os elementos possuem valores de 0 ou 1 e o cromossomo que tiver maior somatório de seus elementos possui melhor avaliação. Dessa forma, a técnica de *streaming* foi utilizada para fazer o somatório desses pontos e foram instanciados 16 *kernels*, sendo cada um responsável por avaliar cada indivíduo. O código final é apresentado no Apêndice A.2 e A.3. O tempo de execução é apresentado na Figura 27.

O Vitis apresenta apenas o gráfico da simulação do código no FPGA, calculando apenas o tempo otimizado pelo uso do FPGA, uma vez que o restante da execução é feita pela CPU. Nesse caso, foi executada a função em 3007 μs apenas, enquanto, que utilizando a CPU, obteve-se um tempo de aproximadamente 6069 μs. O código completo foi executado em 25009 μs e 21947 μs, para a CPU e o FPGA, respectivamente. O resultado é satisfatório, uma vez que foram utilizados poucos métodos de otimização pela aceleração de *hardware* e foi paralelizada apenas uma função do código principal. O comparativo é mostrado na Tabela 8.

	Tempo de execução (μs)		Ganho
	CPU	Com FPGA	
Função de avaliação	6069	3007	50,45%
Código completo	25009	21947	12,24%

Tabela 8 – Comparativo das simulações do AG

Figura 27 – Simulação AG



Fonte: Próprio autor.

Capítulo 5

Conclusões

Analisando os resultados obtidos, conclui-se que a inserção de GDs e VEs na rede elétrica de distribuição de 15 barras pode causar impacto na mesma. Nota-se que houve um aumento tanto no custo de 4,3% e nas perdas de 8,5%, quando analisados os Casos 1 e 2. Em contrapartida, a adição de GDs reduziu o custo do sistema em 19,42% e de suas perdas em 66,7%, no Caso 3, enquanto que no Caso 4, houve um aumento de 12,19% no custo e uma redução de 63,7% das perdas, com a adição de VEs, em relação ao caso 1. É importante ressaltar que a otimização do despacho econômico dos GDs e do carregamento dos VEs devem levar em consideração não só os limites da rede, previsto em normas, mas também das limitações desses componentes, de forma a preservá-los. Assim, nota-se que a aplicação de técnicas de otimização, tais como os AG, podem resultar em diminuição de custos e perdas no sistema elétrico. Para continuidade dessa pesquisa, pretende-se, no futuro, realizar melhorias no algoritmo proposto, como melhorar seu tempo de convergência e adaptá-lo para receber uma rede de maior complexidade. Além disso, pretende-se reduzir a variação de tempo que foi aplicada nesse estudo e assim aplicar um simulador em tempo real. O simulador possuiria a vantagem de se adaptar a mudanças de cargas instantâneas, como novas conexões de VEs não previstas.

Em relação à utilização da técnica de HIL utilizando o FPGA e o Vitis, foram apresentados alguns dos métodos do paralelismo do *software* e aplicá-las ao *hardware*, observando melhorias consideráveis. A contrapartida é que a modelagem possui um grau de complexidade mais elevado do que a modelagem tradicional do problema e o *software* que, além de possuir pré-requisitos elevados em comparação a um computador tradicional, também não possui uma instalação ou utilização trivial, sendo necessário uma leitura de seu manual de forma detalhada. A técnica de *streaming* se mostrou relativamente boa para essa aplicação, porém é recomendada que seja utilizada em casos de dados com muitos elementos, já que em casos de paralelismo utilizando novas instâncias de *kernels*, foi possível verificar que há certas interrupções inicialmente, porém depois é estabilizado depois de alguns microssegundos.

Verificou-se que é importante, durante a modelagem do paralelismo, priorizar as operações que possuem um tempo de elevado de execução e passá-las para o FPGA, pois dessa forma é possível reduzir significativamente esse tempo. Notou-se que as operações em *loop*, geralmente são mais fáceis de serem paralelizáveis e são as que despendem de

grande parte do tempo de processamento e por isso, é aconselhado que o foco inicial seja que essas etapas sejam enviadas aos FPGA de forma a paralelizá-las.

No experimento de modelagem da AG para executar parte no FPGA, obteve-se um ganho de execução relativamente bom, de 50,45% mais rápido em relação a execução da função e, em relação ao tempo total de execução, teve um aumento de aproximadamente 12,24%, apenas aplicando os métodos em uma operação. Em caso de aplicação do método em outras etapas, acredita-se que haverá um ganho maior. Considerando que a estrutura dos AG são semelhantes, ou seja, as operações realizadas nos cromossomos utilizam estruturas de *loop*, que em geral são altamente paralelizáveis, é possível que fosse obtido uma melhora de desempenho caso tivesse sido aplicada ao problema de coordenação de carregamento de VE.

5.1 SUGESTÕES PARA TRABALHOS FUTUROS

Após realizar a pesquisa, foram propostos diversas sugestões para trabalhos futuros. Primeiramente, como forma de obter melhores resultados dos AG, é interessante que outros algoritmos sejam aplicados em conjunto com os AG, pois esse mostrou um resultado interessante para tornar o problema factível, porém para obter resultado muito mais otimizados, os AG necessitam de um tempo muito grande. Para isso, é recomenda-se utilizar algum algoritmo de otimização clássico após verificar que a diferença de melhoria começa a estagnar nos AG. Outra hipótese, seria restringir mais a coordenação, apenas gerando os valores de carregamento dentro de uma rede com GDs com potências pré-definidas, pois dessa forma é possível resultar em uma convergência mais veloz, possibilitando aplicar o algoritmo em redes maiores e com mais VE.

No escopo do HIL e utilização de *software* de HLS, como o Vitis, existem muitas possibilidades para pesquisa. A primeira seria dar continuidade a essa pesquisa e aplicar as técnicas de paralelismo nos AG e obter um resultado mais rápido que foi obtido sem essas técnicas. Mais especificamente, aplicar o método de *streaming* e múltiplas CUs em outras funções, como *crossover*, aplicando duas *streams* de entrada a uma CU e gerando uma *stream* de saída que seria o cromossomo filho, dividindo entre cada *kernel* instanciado, um par de cromossomos pais.

Algumas dificuldades poderiam surgir, devido a aplicações de estruturas condicionais aos *kernels*, uma vez que os dados geralmente são preferivelmente alinhados e sem um desvio da lógica, o que ocorre nas estruturas condicionais. Entretanto, existem outros métodos de paralelizar funções de *loop* como, por exemplo, a técnica de *unrolling*, que consiste em transformar as estruturas de repetição (como, por exemplo, um *for*) em operações individuais dentro de um mesmo *clock*, criando instâncias separadas para cada iteração, para executar em paralelo (XILINX, 2022a). Esse método consiste em

aplicar o *trade-off* entre espaço e tempo, citados no Capítulo 2, permitindo que sejam feitas diversas operações simultaneamente em um ou mais *clocks*, porém utilizando-se mais espaço da memória. Além disso, ainda existe a possibilidade de, em caso de vetores muito grandes e poucos cromossomos na população, particionar o vetor em mais partes, processando em paralelo cada uma dessas partes.

Existem outras diretrizes aplicáveis nos códigos que permitem um melhor aproveitamento do *hardware* utilizado. Além dos previamente citados e utilizados, outra técnica seria utilizar o benefício do *Burst-Read-Write*, que consistem em passar um número maior de dados entre o *host* e o *kernel*, reduzindo os tempos de leitura e escrita na memória global (XILINX, 2022a). Dentro dessas opções, seria interessante fazer um comparativo entre cada método e para cada etapa dos AG, avaliando a factibilidade de cada um e o seu desempenho.

Por fim, vale ressaltar que ainda existe a possibilidade de se utilizar múltiplos *hardwares* de FPGA, o que permitiria resolver problemas maiores com maior agilidade, aumentando a paralelidade do sistema.

REFERÊNCIAS

ALVAREZ, J. G.; GONZÁLEZ, M. Á.; VELA, C. R.; VARELA, R. Electric vehicle charging scheduling by an enhanced artificial bee colony algorithm. **Energies**, Multidisciplinary Digital Publishing Institute, v. 11, n. 10, p. 2752, 2018.

ASANO, S.; MARUYAMA, T.; YAMAGUCHI, Y. Performance comparison of fpga, gpu and cpu in image processing. In: IEEE. **2009 international conference on field programmable logic and applications**. [S.l.], 2009. p. 126–131.

BALEN, G.; REIS, A. R.; PINHEIRO, H.; SCHUCH, L. Estação de carregamento rápido com elemento armazenador de energia e filtro ativo de harmônicos para veículos elétricos. **Revista Eletrônica de Potência**, v. 24, n. 1, p. 95–106, 2019.

BARAN, M. E.; WU, F. F. Network reconfiguration in distribution systems for loss reduction and load balancing. **IEEE Power Engineering Review**, IEEE, v. 9, n. 4, p. 101–102, 1989.

BÉLANGER, J.; VENNE, P.; PAQUIN, J.-N. The what, where and why of real-time simulation. **Planet Rt**, v. 1, n. 1, p. 25–29, 2010.

BOULANGER, A. G.; CHU, A. C.; MAXX, S.; WALTZ, D. L. Vehicle electrification: Status and issues. **Proceedings of the IEEE**, IEEE, v. 99, n. 6, p. 1116–1138, 2011.

CARPENTIER, J. Contribution a l'etude du dispatching economique. **Bulletin de la Societe Francaise des Electriciens**, v. 3, n. 1, p. 431–447, 1962.

CESPEDES, R. New method for the analysis of distribution networks. **IEEE Transactions on Power Delivery**, IEEE, v. 5, n. 1, p. 391–396, 1990.

CHEN, F.; CHEN, Z.; DONG, H.; YIN, Z.; WANG, Y.; LIU, J. Research on the influence of electric vehicle multi-factor charging load on a regional power grid. In: IEEE. **2018 10th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)**. [S.l.], 2018. p. 163–166.

FARUQUE, M. O.; STRASSER, T.; LAUSS, G.; JALILI-MARANDI, V.; FORSYTH, P.; DUFOUR, C.; DINAVAHI, V.; MONTI, A.; KOTSAMPOPOULOS, P.; MARTINEZ, J. A. et al. Real-time simulation technologies for power systems design, testing, and analysis. **IEEE Power and Energy Technology Systems Journal**, IEEE, v. 2, n. 2, p. 63–73, 2015.

FOURER, R.; GAY, D. M.; KERNIGHAN, B. W. A modeling language for mathematical programming. **Management Science**, INFORMS, v. 36, n. 5, p. 519–554, 1990.

GARCIA-OSORIO, V. A.; RUEDA-MEDINA, A. C.; MELO, J.; PADILHA-FELTRIN, A. Optimal charging of electric vehicles considering constraints of the medium voltage distribution network. In: IEEE. **2013 IEEE PES Conference on Innovative Smart Grid Technologies (ISGT Latin America)**. [S.l.], 2013. p. 1–7.

GARCIA, P.; BHOWMIK, D.; STEWART, R.; MICHAELSON, G.; WALLACE, A. Optimized memory allocation and power minimization for fpga-based image processing. **Journal of Imaging**, MDPI, v. 5, n. 1, p. 7, 2019.

GCC, t. G. C. C. **GCC Documentation**. 2022. Disponível em: <https://gcc.gnu.org/>.

HAJFOROOSH, S.; MASOUM, M. A.; ISLAM, S. M. Real-time charging coordination of plug-in electric vehicles based on hybrid fuzzy discrete particle swarm optimization. **Electric Power Systems Research**, Elsevier, v. 128, p. 19–29, 2015.

HANGÜN, B. **Performance Evaluation of a Parallel Image Enhancement Technique for Dark Images on Multithreaded Cpu and Gpu Architectures**. Tese (Doutorado) — Marmara Universitesi (Turkey), 2021.

HAUPT, R. L.; HAUPT, S. E. **Practical genetic algorithms**. [S.l.]: John Wiley & Sons, 2004.

KARMARKAR, N. A new polynomial-time algorithm for linear programming. In: **Proceedings of the sixteenth annual ACM symposium on Theory of computing**. [S.l.: s.n.], 1984. p. 302–311.

LESAJA, G. Introducing interior-point methods for introductory operations research courses and/or linear programming courses. **Open Operational Research Journal**, v. 3, p. 1, 2009.

MEDINA, A. C. R. Simutere: Simulador em tempo real de sistemas de energia. In: UNIVERSIDADE FEDERAL DO ESPíRITO SANTO. [S.l.], 2019.

MORELLO, R.; RIENZO, R. D.; RONCELLA, R.; SALETTI, R.; BARONTI, F. Hardware-in-the-loop platform for assessing battery state estimators in electric vehicles. **IEEE Access**, IEEE, v. 6, p. 68210–68220, 2018.

RAZZAGHI, R.; PAOLONE, M.; RACHIDI, F. A general purpose fpga-based real-time simulator for power systems applications. In: IEEE. **IEEE PES ISGT Europe 2013**. [S.l.], 2013. p. 1–5.

ROINILA, T.; MESSO, T.; LUHTALA, R.; SCHARRENBERG, R.; JONG, E. C. de; FABIAN, A.; SUN, Y. Hardware-in-the-loop methods for real-time frequency-response measurements of on-board power distribution systems. **IEEE Transactions on Industrial Electronics**, IEEE, v. 66, n. 7, p. 5769–5777, 2018.

RUEDA-MEDINA, A. C.; FRANCO, J. F.; RIDER, M. J.; PADILHA-FELTRIN, A.; ROMERO, R. A mixed-integer linear programming approach for optimal type, size and allocation of distributed generation in radial distribution systems. **Electric power systems research**, Elsevier, v. 97, p. 133–143, 2013.

SILVA, F. Z. da; RUEDA-MEDINA, A. C. Um método híbrido de otimização para despacho econômico e alocação de gds e estações de carregamento de veículos elétricos. **Simpósio Brasileiro de Sistemas Elétricos-SBSE**, v. 1, n. 1, 2020.

SINGH, K. V.; BANSAL, H. O.; SINGH, D. Hardware-in-the-loop implementation of anfis based adaptive soc estimation of lithium-ion battery for hybrid vehicle applications. **Journal of Energy Storage**, Elsevier, v. 27, p. 101124, 2020.

WACHTER, A. **An interior point algorithm for large-scale nonlinear optimization with applications in process engineering**. Tese (Doutorado) — Carnegie Mellon University, 2002.

WANG, Y.; HUANG, Y.; WANG, Y.; ZENG, M.; LI, F.; WANG, Y.; ZHANG, Y. Energy management of smart micro-grid with response loads and distributed generation considering demand response. **Journal of cleaner production**, Elsevier, v. 197, p. 1069–1083, 2018.

WANG, Y.; SYED, M. H.; GUILLO-SANSANO, E.; XU, Y.; BURT, G. M. Inverter-based voltage control of distribution networks: A three-level coordinated method and power hardware-in-the-loop validation. **IEEE Transactions on Sustainable Energy**, IEEE, v. 11, n. 4, p. 2380–2391, 2019.

WILSON, P. **Design recipes for FPGAs: using Verilog and VHDL**. [S.l.]: Newnes, 2015.

XILINX. **Vivado Design Suite User Guide (UG901)**. Xilinx, 2019. Disponível em: <https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis>.

XILINX. **Vitis AI: Support for Zynq-7000 devices**. 2021. Disponível em: <https://support.xilinx.com/s/article/76742>.

XILINX. **Vitis High-Level Synthesis User Guide (UG1399)**. Xilinx, 2022. Disponível em: <https://docs.xilinx.com/viewer/book-attachment/u1ha7A~FnJAUGn1TvNNmSQ/Cs1ywAU1szvAc27sj6jP5g>.

XILINX. **What is an FPGA**. 2022. Disponível em: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>.

YAN, L.; XINSHOU, T.; JUAN, H.; ZHANKUI, Z.; HUI, Y.; XIONG, X.; MING, W.; WENYUAN, M. Study on distributed generation to improve stability of distribution network based on virtual synchronous generator technology. In: IEEE. **2019 IEEE 8th International Conference on Advanced Power System Automation and Protection (APAP)**. [S.l.], 2019. p. 98–103.

YILMAZ, M.; KREIN, P. T. Review of battery charger topologies, charging power levels, and infrastructure for plug-in electric and hybrid vehicles. **IEEE transactions on Power Electronics**, IEEE, v. 28, n. 5, p. 2151–2169, 2012.

ZHU, J. **Optimization of power system operation**. [S.l.]: John Wiley & Sons, 2015.

APÊNDICE A

Códigos Implementados

A.1 ADIÇÃO DE VETORES (ZEDBOARD ZYNQ-7000)

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xtime_l.h"

// #pragma GCC optimize("O1")

int main()
{
   init_platform();
  XTime tStart, tEnd;
  int size = 1000;
  int a[size];
  int b[size];
  int c[size];

  for(int i = 0; i <= size; i++) {
    a[i] = i;
    b[i] = i;
  }

   XTime_GetTime(&tStart);
   //#pragma GCC push_options
  for(int i = 0; i <= size; i++) {
    c[i] = a[i] + b[i];
  }
  //#pragma GCC pop_options
  XTime_GetTime(&tEnd);

   printf("Output took %llu clock cycles.\n", 2*(tEnd - tStart));
     printf("Output took %.2f us.\n",
```

```
            1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));
    xil_printf("c[1000]: a[1000] + b[1000] => 1000 + 1000 = %d\n",
       c[size]);
    cleanup_platform();
    return 0;
}
```

A.2 *HOST* DO AG

```
#include "xcl2.hpp"
#include <vector>

#define DATA_SIZE 4 * 4096
#define num_cu 16
#define generations 10
#define tournament_size num_cu/2
#define pop_size num_cu

void crossover(std::vector<int, aligned_allocator<int>> dad,
    std::vector<int, aligned_allocator<int>> mom, std::vector<int>
    &child) {
  int co_point = rand() % DATA_SIZE;

  for (int i = 0; i < co_point; i++) {
    child.push_back(dad[i]);
  }

  for (int i = co_point; i < DATA_SIZE; i++) {
    child.push_back(mom[i]);
  }
}

int getBest(std::vector<int> all_scores) {
  int best_index = 0;
  int best_score = 0;

  for (int i = 0; i < pop_size; i++){
    int score = all_scores[i];
    if (score > best_score) {
      best_index = i;
      best_score = score;
```

```
      }
    }
    return best_index;
}

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " <XCLBIN File>" <<
            std::endl;
        return EXIT_FAILURE;
    }

    auto binaryFile = argv[1];

    // Allocate Memory in Host Memory
    cl_int err;
    cl::Context context;
    cl::CommandQueue q;
    std::vector<cl::Kernel> krnl_adder(num_cu);
    size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
    std::vector<int, aligned_allocator<int> > source_input(DATA_SIZE);
    std::vector<std::vector<int, aligned_allocator<int>>>
        source_hw_results(pop_size, std::vector<int,
        aligned_allocator<int>>(DATA_SIZE));

    // OPENCL HOST CODE AREA START
    auto devices = xcl::get_xil_devices();

    // Create Program and Kernel
    auto fileBuf = xcl::read_binary_file(binaryFile);
    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
    bool valid_device = false;
    for (unsigned int i = 0; i < devices.size(); i++) {
        auto device = devices[i];
        // Creating Context and Command Queue for selected Device
        OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr,
            nullptr, &err));
        OCL_CHECK(err, q = cl::CommandQueue(context, device,
            CL_QUEUE_PROFILING_ENABLE |
                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));
```

```
        std::cout << "Trying to program device[" << i << "]: " <<
            device.getInfo<CL_DEVICE_NAME>() << std::endl;
        cl::Program program(context, {device}, bins, nullptr, &err);
        if (err != CL_SUCCESS) {
            std::cout << "Failed to program device[" << i << "] with
                xclbin file!\n";
        } else {
            std::cout << "Device[" << i << "]: program successful!\n";
            for (int i = 0; i < num_cu; i++) {
              OCL_CHECK(err, krnl_adder[i] = cl::Kernel(program,
                 "adder", &err));
            }
            valid_device = true;
            break; // we break because we found a valid device
        }
    }
    if (!valid_device) {
        std::cout << "Failed to program any device found, exit!\n";
        exit(EXIT_FAILURE);
    }


    // Start of the GA
    std::vector<cl::Buffer> buffer_input(pop_size);
    std::vector<cl::Buffer> buffer_output(pop_size);
    std::vector<std::vector<int, aligned_allocator<int>>>
        chromosomes(pop_size, std::vector<int,
        aligned_allocator<int>>(DATA_SIZE));

    for (int i = 0; i < pop_size; i++) {
      for (int j = 0; j < DATA_SIZE; j++) {
        chromosomes[i][j] = rand() % 2;
      }
    }

    for (int g = 0; g < generations; g++) {
      std::cout << "Generation " << g+1;

    ////////////////////// start of eval function ////////////////////
      for (int i = 0; i < pop_size; i++) {
        // Allocate Buffer in Global Memory
```

```
      OCL_CHECK(err, buffer_input[i] = cl::Buffer (context,
           CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
           chromosomes[i].data(), &err));
      OCL_CHECK(err, buffer_output[i] = cl::Buffer(context,
           CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
              vector_size_bytes,
           source_hw_results[i].data(), &err));
      // Copy input data to device global memory
      OCL_CHECK(err, err =
         q.enqueueMigrateMemObjects({buffer_input[i]}, 0 /* 0 means
         from host*/));
    }

    for (int i = 0; i < pop_size; i++) {
      int size = DATA_SIZE;
      // Set the Kernel Arguments
      int narg = 0;
      OCL_CHECK(err, err = krnl_adder[i].setArg(narg++,
         buffer_input[i]));
      OCL_CHECK(err, err = krnl_adder[i].setArg(narg++,
         buffer_output[i]));
      OCL_CHECK(err, err = krnl_adder[i].setArg(narg++, size));
    }

    for (int i=0; i < pop_size; i++) {
      // Launch the Kernel
      OCL_CHECK(err, err = q.enqueueTask(krnl_adder[i]));
    }

    for (int i = 0; i < pop_size; i++) {
      // Copy Result from Device Global Memory to Host Local Memory
      OCL_CHECK(err, err =
         q.enqueueMigrateMemObjects({buffer_output[i]},
         CL_MIGRATE_MEM_OBJECT_HOST));
    }

    q.finish();
  //////////////////// end of eval function ///////////////////////

    std::vector<int> all_scores;
    for (int i = 0; i < pop_size; i++) {
```

```
        all_scores.push_back(source_hw_results[i][DATA_SIZE-1]);
      }

      int best_index = getBest(all_scores);

      for (int i = 0; i < tournament_size; i++) {
        int dad_index = best_index;
        int mom_index;
        do {
          mom_index = rand() % pop_size;
        }while(dad_index == mom_index);

        std::vector<int> child;
        crossover(chromosomes[dad_index], chromosomes[mom_index],
           child);

        for (int j = 0; j < DATA_SIZE; j++) {
          chromosomes[mom_index][i] = child[i];
        }

      }
      std::cout << " -- Done G" << g+1 << std::endl;

    }
    return 0;
}
```

A.3 *KERNEL* DO AG

```
#include <ap_int.h>
#include <hls_stream.h>

#define DATA_SIZE 4 * 4096

// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

// Read Data from Global Memory and write into Stream inStream
static void read_input(unsigned int* in, hls::stream<unsigned int>&
    inStream, int size) {
// Auto-pipeline is going to apply pipeline to this loop
```

```
mem_rd:
   for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
      // Blocking write command to inStream
      inStream << in[i];
   }
}

// Read Input data from inStream and write the result into outStream
static void compute_add(hls::stream<unsigned int>& inStream,
   hls::stream<unsigned int>& outStream, int size) {
// Auto-pipeline is going to apply pipeline to this loop
execute:
   for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
      // Blocking read command from inStream and Blocking write
         command
      // to outStream
      outStream << (inStream.read());
   }
}

// Read result from outStream and write the result to Global Memory
static void write_result(unsigned int* out, hls::stream<unsigned
   int>& outStream, int size) {
// Auto-pipeline is going to apply pipeline to this loop
  int sum = 0;
mem_wr:
   for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
      // Blocking read command to inStream
     sum = sum + outStream.read();
      out[i] = sum;
   }
}

extern "C" {
/*
   Vector Eval Kernel Implementation using dataflow
   Arguments:
      in  (input) --> Input Vector
```

```
      out (output) --> Output Vector
      size (input) --> Size of Vector in Integer
  */
   void adder(unsigned int* in, unsigned int* out, int size) {
      static hls::stream<unsigned int> inStream("input_stream");
      static hls::stream<unsigned int> outStream("output_stream");
   #pragma HLS STREAM variable = inStream depth = 32
   #pragma HLS STREAM variable = outStream depth = 32

   #pragma HLS dataflow
      // dataflow pragma instruct compiler to run following three
         APIs in parallel
      read_input(in, inStream, size);
      compute_add(inStream, outStream, size);
      write_result(out, outStream, size);
   }
}
```