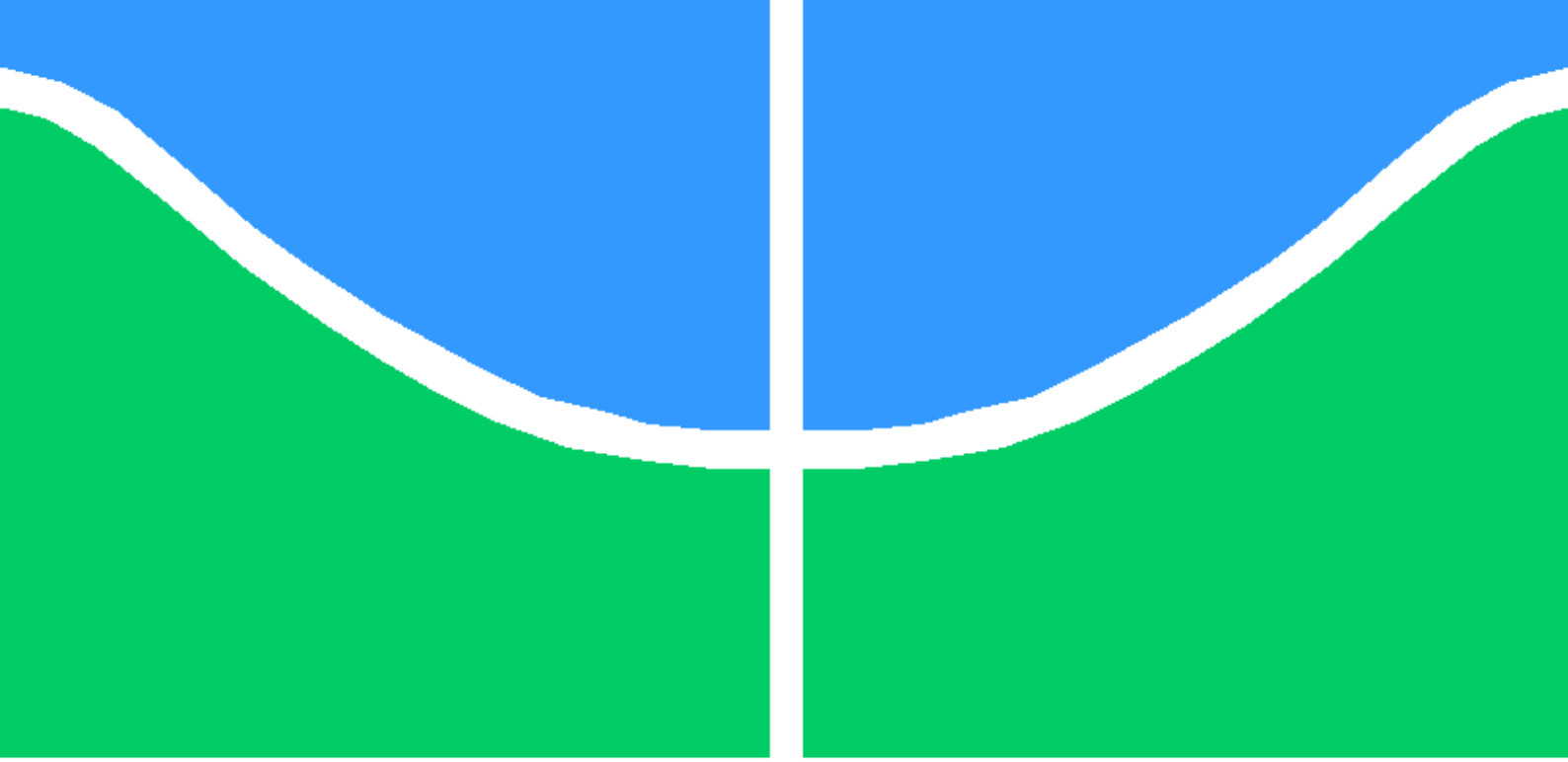


MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA

LUIZ AUGUSTO DOS SANTOS PIRES

DISSERTAÇÃO DE GRADUAÇÃO EM ENGENHARIA DE REDES DE COMUNICAÇÃO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA

LUIZ AUGUSTO DOS SANTOS PIRES

Orientador: PROF. DR. WILLIAM FERREIRA GIOZZA, ENE/UNB
Coorientador: PROF. DR. ALEXANDRE SOLON NERY, ENE/UNB

DISSERTAÇÃO DE GRADUAÇÃO EM ENGENHARIA DE REDES DE COMUNICAÇÃO

PUBLICAÇÃO PPGENE.DM - XXX/AAAA
BRASÍLIA-DF, 29 DE ABRIL DE 2022.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE REDES DE COMUNICAÇÃO

MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA

LUIZ AUGUSTO DOS SANTOS PIRES

DISSERTAÇÃO DE GRADUAÇÃO ACADÊMICO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE GRADUAÇÃO EM ENGENHARIA DE REDES DE COMUNICAÇÃO.

APROVADA POR:

Prof. Dr. William Ferreira Giozza, ENE/UnB
Orientador

Prof. Dr. Georges Daniel Amvame Nze, ENE/UnB
Examinador interno

Prof. Dr. Alexandre Solon Nery, ENE/UnB
Examinador interno

BRASÍLIA, 29 DE ABRIL DE 2022.

FICHA CATALOGRÁFICA
LUIZ AUGUSTO DOS SANTOS PIRES
MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA
2022xv, 147p., 201x297 mm
(ENE/FT/UnB, Graduação, Engenharia de Redes de Comunicação, 2022)
Dissertação de Graduação - Universidade de Brasília
Faculdade de Tecnologia - Departamento de Engenharia Elétrica

REFERÊNCIA BIBLIOGRÁFICA

LUIZ AUGUSTO DOS SANTOS PIRES (2022) MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA. Dissertação de Graduação em Engenharia de Redes de Comunicação, Publicação xxx/AAAA, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 147p.

CESSÃO DE DIREITOS

AUTOR: LUIZ AUGUSTO DOS SANTOS PIRES
TÍTULO: MEDIDAS DE CENTRALIDADE EM GRAFOS IMPLEMENTADAS EM FPGA.
GRAU: Graduação ANO: 2022

_____

LUIZ AUGUSTO DOS SANTOS PIRES
SGAS 905 BLOCO D APT 219

Agradecimentos

Agradeço a Deus por ter me dado a sabedoria e paciência necessária para escrever o trabalho e por ter colocado as pessoas certas para me ajudar no decorrer do trabalho.

Agradeço aos meus familiares por terem me incentivado e tranquilizado nos momentos mais difíceis do trabalho.

Agradeço ao meu orientador, Prof. William Ferreira Giozza, por todo apoio e paciência durante a realização do trabalho.

Agradeço ao Prof. Alexandre Solon Nery por ter me ajudado a escolher o objeto de pesquisa e ter tirado minhas dúvidas a respeito dos temas do trabalho sempre que precisava.

Agradeço ao meu amigo, Lucas Martins Alves, por sua disponibilidade e por ter me ajudado em dúvidas que tive durante o trabalho.

Resumo

O presente trabalho busca implementar o algoritmo *Betweenness Centrality* (BC) em um *Field Programmable Gate Array* (FPGA), de forma mais especifica o algoritmo de Brandes tendo Dijkstra como núcleo para suportar grafos com pesos. O intuito foi identificar a viabilidade de sua implementação em FPGAs, bem como os ganhos no tempo de inferência do algoritmo e avaliar o consumo energético do circuito sintetizado. Para realizar a implementação usou-se o compilador Vitis HLS, que permite usar programação em C, C++ para gerar o modelo *Register Transfer Language* (RTL) facilitando o processo de implementação de algoritmos complexos em FPGAs.

Abstract

The present work seeks to implement the Betweenness Centrality algorithm in an Field Programmable Gate Array (FPGA), more specifically the Brandes algorithm having Dijkstra as the core to support weighted graphs. The aim was to identify the feasibility of its implementation in FPGAs, as well as the gains in the algorithm's inference time and to evaluate the energy consumption of the synthesized circuit. To carry out the implementation, the Vitis HLS compiler was used, which allows using programming in C, C++ to generate the Register Transfer Language (RTL) model, facilitating the process of implementing complex algorithms in FPGAs.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS FONTE

LISTA DE TERMOS E SIGLAS

APSP	*All Pairs Shortest Path*
ARM	*Advanced RISC Machines*
AXI	*Advanced eXtensible Interface*
BC	*Betwenness Centrality*
BD	*Block Design*
BFS	*Breadth-First Search*
BRAM	*Block Random Access Memory*
CPU	*Central Processing Unit*
DDR	*Double Data Rate*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processing*
FF	*Flip-Flops*
FPGA	*Field Programmable Gate Array*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HLS	*High Level Synthesis*
HP	*High Performance*
IEEE	*Institute of Electrical and Electronic Engineers*
IP	*Intellectual Property*
LUT	*LookUp Table*
MIO	*Multiplexed Input/Output*

MPSoCs	*Multiprocessor system on a chip*
PL	*Programmable Logic*
RAW	*Read After Write*
RTL	*Register Transfer Language*
SAIF	*Switching Activity Interchange Format*
SCU	*Specialized Container Unit*
SDRAM	*Synchronous Dynamic Random Access Memory*
SMU	*Specialized Method Unit*
SOC	*System on a chip*
SSH	*Secure Shell*
SSSP	*Single Source Shortest Path*
TNS	*Total Negative Slack*
URAM	*Ultra Random Access Memory*
WAR	*Write After Read*
WNS	*Worst Negative Slack*

Capítulo 1

Introdução

1.1 Motivação

O processamento de algoritmos em grafos tornou-se importante para muitas áreas, como *machine learning* [1], cybersegurança [1] e análises em redes sociais [2]. Implementações usando arquiteturas heterogêneas de CPU (*Central Processing Unit*) / GPU (*Graphics Processing Unit*) buscam atingir melhor desempenho temporal a partir de um paralelismo proporcionado pelas GPUs e por múltiplas *threads* operando concomitantemente. Tais implementações por vezes podem reduzir o tempo de inferência do algoritmo de meses para dias no caso de grafos com milhares de arestas como mostrado em [3], em que mais especificamente é implementado o algoritmo *Betwenness Centrality* (BC) para extrair medidas de centralidade dos grafos. No caso, este algoritmo permite inferir quais são os nós mais influentes em um grafo.

Ao analisar trabalhos de segurança de rede na área de detecção de *Botnets*, percebeu-se que algoritmos que extraem medidas de centralidade, como *Betwenness Centrality*, estão sendo usados como métrica para verificar ataques deste tipo. Existem trabalhos que implementam tal algoritmo em arquiteturas heterogêneas de CPU/GPU como [3] e [4] conseguindo com isso resultados promissores. Porém pouco se discute sobre sua viabilidade de implementação em sistemas que usam microprocessadores ARM (*Advanced RISC Machines*), tais como FPGAs que incorporam o microprocessador em torno da lógica programável de seus chips reconfiguráveis.

1.2 Objetivos e Contribuições

A fim de investigar a viabilidade de se implementar algoritmos de Centralidade em FPGAs, pretende-se implementar o algoritmo BC em um FPGA Pynq-z1 (ZYNQ XC7Z020-1CLG400C). Após implementar o circuito, chamado comumente de Propriedade intelectual (IP), na lógica programável (PL) do FPGA, pretende-se comparar o desempenho do proces-

sador ARM operando sozinho com o desempenho dele usando um co-processador sintetizado no PL.

Escolheu-se o BC para realizar esta validação por se tratar de um algoritmo que utiliza em seu núcleo algoritmos SSSP (caminhos mais curtos de fonte única) como Dijkstra e Bellman-Ford. Estes são algoritmos que tem bastante suporte literário e são relativamente simples de se implementar. Além disso o BC vem sendo usado em trabalhos de segurança de rede como em [1]. A implementação do algoritmo será feita com auxílio do Vitis HLS Compiler. Esta ferramenta recebe na entrada um código em C++ e gera um código em HDL (*Hardware Description Language*) na saída, que descreve a arquitetura do circuito no modelo RTL (*Register Transfer Language*) .

As contribuições deste trabalho estão em melhorar o acervo de algoritmos de centralidade implementados em FPGAs e sugerir de forma detalhada as modificações necessárias no algoritmo de Dijkstra (que permite processamento em grafos com pesos) para operar no BC. Para tanto, foram feitos pseudocódigos (baseados no pseudocódigo de [5] para grafos sem peso) e explicações ao longo do trabalho para mostrar como os algoritmos devem operar em conjunto. Para implementar BC usou-se o algoritmo de Brandes [5]. Conseguiu-se ao final montar uma arquitetura que consegue obter valores melhores de tempo de processamento no FPGA ao se usar o co-processador operando junto ao ARM.

1.3 Organização do trabalho

O trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os conceitos fundamentais em teoria dos grafos, bem como a descrição de diversos métodos e algoritmos para se computar o algoritmo BC. Também são descritos trabalhos relacionados levando em conta implementações realizadas em FPGAs; o Capítulo 3 descreve o método proposto neste trabalho que consiste na justificativa de escolha do algoritmo para computar o BC, na confecção de um código em C++, nas etapas de otimização e síntese para gerar o código HDL no Vitis HLS, nas etapas de implementação e geração do arquivo .bit, necessárias para o mapeamento do circuito no FPGA e para que a arquitetura possa ser construída no PL, usando o Vivado e por último a implementação de um código em Python para comunicação entre PS e PL no FPGA; o Capítulo 4 apresenta os resultados obtidos pelo método proposto levando em conta o tempo e consumo energético do FPGA; o Capítulo 5 compara o resultado com os obtidos em outros trabalhos feitos em FPGAs e discute melhorias possíveis de serem feitas em trabalhos futuros. Por último o Capítulo final apresenta as conclusões do trabalho.

Capítulo 2

Referencial Teórico

2.1 Fundamentos

Antes de descrever o algoritmo BC é necessário realizar uma revisão para conceituar os principais elementos dentro da teoria dos grafos.

2.1.1 Teoria dos Grafos

Para entender o funcionamento de algoritmos dentro da teoria dos grafos é necessário antes definir claramente o que é um grafo. Para isto, inspirou-se nas definições e notações dadas por [6].

Para iniciar, considere V um conjunto qualquer. Neste caso $V^{(2)}$ será o conjunto de todos os pares não ordenados de elementos de V. Se V tem n elementos então $V^{(2)}$ tem $\binom{n}{2} := \frac{n(n-1))}{2}$ elementos. Assim $V^{(2)}$ terá forma $\{v, w\}$, sendo v e w dois elementos distintos de V.

Um grafo consiste de um par G = (V,E) sendo que V é um conjunto arbitrário e E é um subconjunto de $V^{(2)}$ como explica [6]. Dito isto os elementos de V são chamados de vértices e os de E são chamados de arestas. O número de vértices e arestas será denotado no trabalho como |V| e |E|, respectivamente.

Dada esta definição de grafo também é importante definir algumas nomenclaturas que serão utilizadas. Usando ainda os conceitos em [6] é dito que se $\{v, w\}$ é uma aresta, então os vértices v e w são vizinhos ou adjacentes. Também é definido que a aresta $\{v, w\}$ incide em v e em w e que v e w são as pontas da aresta.

Outro conceito importante a se definir é o digrafo que é um grafo com arestas direcionadas. Com isso $V^{(2)}$ é o conjunto de todos os pares ordenados de elementos de V. Assim as arestas $\{v, w\}$ e $\{w, v\}$ são diferentes. O conjunto de arestas para um grafo direcionado de n vértices pode conter até $A_{n,2} = n(n-1)$ elementos.

No presente trabalho ao se referir a palavra grafo está se falando tanto de grafos dire-

cionados quanto não direcionados. Caso determinado conceito aplique-se a somente a um destes então irá se especificar se o grafo é direcionado ou não.

Esclarece-se que o grau de um vértice v, $deg(v)$, é o número de arestas incidentes com v. Se $deg(v)$ = 0 significa que o vértice está isolado.

Existem várias representações para os grafos. A representação utilizada no trabalho será a chamada matriz de adjacência. Seguindo a definição de [6] a matriz de adjacências de um grafo é uma matriz A com linhas e colunas indexadas pelos vértices do grafo G, V(G), tal que A[u, v] = 1 se $\{u, v\}$ ϵ $E(G)$, que é o conjunto de arestas de G, e igual a 0 em caso contrário. Quando um grafo é não direcionado A[u, v] = A[v, u], mas para um grafo direcionado isto pode não ser verdade.Os grafos também podem ser representados por uma lista de adjacências, que especifica todos os vértices adjacentes a cada vértice no grafo. Esta abordagem pode ser útil caso se esteja lidando com um grafo esparso [7].

Defini-se ,por fim, o conceito de grafo esparso e grafo denso.Para grafos densos o número de arestas é da mesma ordem que o quadrado do número de vértices ,como, $\frac{|V|^2}{2}$ ou $\frac{|V|^2}{8}$. Enquanto que em um grafo esparso o número de arestas é da mesma ordem do número de vértices, por exemplo, $|V|$ ou $10|V|$.

2.1.2 Algoritmos de menor Custo

Um passeio em um grafo é dado por uma sequência de vértices, sendo que dois vértices consecutivos(vértices que ligados formam uma aresta) formam um arco do grafo. Um caminho é um passeio sem arcos repetidos. Diz-se que um caminho é simples se este não tem vértices repetidos. É possível que os arcos em um grafo tenham custos arbitrários definidos. Tais definições estão alinhadas segundo [8]. No trabalho em questão sempre que se refere a caminho está se referindo a definição de caminho simples, ou seja, não há repetição de vértices. Por último, o custo de um caminho é a soma dos custos dos arcos no caminho.

Levando em conta estes conceitos, existe um problema comum em teoria dos grafos que é o de calcular caminhos mínimos de um vértice de origem v a um de destino w. Um caminho C é dito mínimo se não existir outro caminho com origem e destino iguais aos de C com custo menor do que o de C como explicado por [8].

Muitas vezes pode-se usar o termo distância entre dois vértices para se referir ao caminho mínimo entre um vértice v de origem e um w de destino.

Nesta seção explora-se algoritmos de caminhos mínimos que estão intimamente ligados a proposta do trabalho. As descrições são feitas de modo a ajudar o leitor a compreender o algoritmo que será proposto.

O algoritmo de Dijkstra é usado para encontrar as distâncias entre um vértice s de origem, chamado raiz, e todos os outros vértices do grafo, sendo classificado como um algoritmo SSSP. Ele consegue resolver este problema somente se as arestas tiverem custos positivos.

2.1.2.1 Dijkstra

Algorithm 1: Dijkstra

Input: (w := adjacency matrix), src
Output: $dist$, $prev$, $VisitedNodes$

```
1  Declare dist, prev, visitedNodes
2  for each vertex v in Graph do
3      dist[v] ← INFINITY
4      pred[v] ← UNDEFINED
5  dist[src] ← 0
6  pred[src] ← src
7  while len(VisitedNodes) != |V| do
8      v = vertex not in visitedNodes with min dist[v]
9      if dist[v] == infinity then
10         break
11     add v to visitedNodes
12     for all vertex u adjacent with v and not in visitedNodes do
13         if dist[u] > dist[v] + w[u,v] then
14             dist[u] ← dist[v] + w[u, v]
15             pred[u] = v
```

O algoritmo 1 mostra um pseudocódigo de Dijkstra. Ele foi feito para ajudar o leitor no entendimento da proposta do trabalho, já que o mesmo algoritmo será modificado nos capítulos posteriores. Em cada iteração **While** (linha 7) do algoritmo, um nó é adicionado a uma árvore, que pode ser obtida a partir de **pred**, que contém os predecessores dos nós no caminho mínimo. Após o término do algoritmo tem-se então uma árvore contendo a origem como raiz e todos os nós conectados a origem, direta ou indiretamente, acessíveis por meio do seu caminho mínimo.

O algoritmo necessita de algumas estruturas de dados para armazenar informações durante as iterações do algoritmo. As principais estruturas são: **dist**, **pred** e **visitedNodes**. A estrutura **dist** geralmente é implementada usando uma fila de prioridade, o que garante melhor eficiência para extrair a menor distância em cada iteração [7]. O array **pred** é usado para armazenar o predecessor de um nó no caminho de menor custo até a origem sendo que **pred[v]** denota o predecessor do nó v no caminho de menor custo. Por último está o array que guarda os nós na ordem em que estão sendo visitados chamado de **visitedNodes**.

A complexidade do algoritmo é de O($|V|^2$), mas pode ser reduzida caso seja usada uma fila de prioridade implementada como uma heap para ordenação dos pesos dos nós em cada iteração do algoritmo, como explica [7]. Implementações usando Fibonacci heap tem complexidade de tempo de $O(|E|+|V|log|V|)$ enquanto ao se utilizar uma Binary heap chega-se em $O(|E|log|V|)$, como observa [9].

2.1.2.2 Algoritmo Floyd–Warshall

Com execução do Floyd–Warshall, diferente do algoritmo de Dijkstra, é possível obter todas os caminhos mínimos possíveis entre quaisquer pares de vértices sendo, portanto, um algoritmo *All Pairs Shortest Path* (APSP) . Este algoritmo também permite a inclusão de pesos negativos, mas sem ciclos negativos.

Neste algoritmo as distâncias mínimas são guardadas em uma estrutura nomeado de **dist**, que precisa guardar as distâncias entre todos os pares de vértices existentes no grafo. Sendo, portanto, um array bidimensional com os vértices como índices. O pseudo-código de Floyd–Warshall é mostrado no Algoritmo 2.

Algorithm 2: Floyd–Warshall algorithm

```
   Input: (w := matriz de adjacência)
   Output: dist
1  Declare dist // |V| x |V| matrix
2  for each edge (u,v) do do
3      dist[u][v] ← w[u, v]
4  for each vertex v do do
5      dist[v][v] ← 0
6  for i from 1 to |V| do
7      for j from 1 to |V| do
8          for k from 1 to |V| do
9              if dist[j][k] > dist[j][i] + dist[i][k] then
10                 dist[j][k] ← dist[j][i] + dist[i][k]
```

O algoritmo Floyd–Warshall constrói a matriz de caminhos mínimos adicionando em cada iteração do laço mais externo, mostrado no Algoritmo 2 na linha 6, um vértice i como possível intermediário caso ele satisfaça a condição: $dist[j][k] > dist[j][i] + dist[1][i]$, ou seja, se existe um caminho de menor custo de j a k passando por i. Se isto for verdade, a distância de j a k é atualizada. Tal embasamento teórico foi retirado de [7].

No pseudo-código mostrado no Algoritmo 2 não se usou uma estrutura para armazenar os vértices no caminho mínimo, embora seja possível fazê-lo. Este procedimento pode ser custoso em termos de memória por necessitar armazenar os predecessores para todos os pares de vértices, ou seja, entre todas as origens e todos os destinos possíveis.

Todos os três laços mostrados no pseudocódigo do Algoritmo 2 são executados |V| vezes, portanto a complexidade do algoritmo é O($|V|^3$).

Este algoritmo é uma boa abordagem para grafos densos, que são quase completos, mas em grafos esparsos não há necessidade de verificar todas as conexões possíveis entre os vértices, podendo ser mais vantajoso usar o método um para todos(como Dijkstra) |V| vezes como explica [7].

2.1.2.3 Algoritmo Bellman–Ford

Bellman-Ford é um algoritmo SSSP(*Single Source Shortest Path*), podendo funcionar com arestas com pesos negativos, porém sem ciclos negativos [7].

Ele funciona fazendo o processo de relaxamento, ou seja, aproxima uma distância que acredita ser a melhor, mas que pode ser substituída no decorrer do algoritmo por outra até que se alcance a solução. No caso do Dijkstra, o relaxamento se dá apenas nas arestas de saída para o nó que acaba de ser considerado visitado (incluído em $VisitedNodes$), enquanto aqui o relaxamento ocorre em todas as arestas do grafo, processo que se repete (n° de vértices -1) vezes até o fim do algoritmo.

A complexidade do algoritmo é de $O(|V||E|)$. Havendo no máximo |V|-1 passes através da sequência de |E| arestas, pois |V|-1 é o maior número de arestas para qualquer caminho como evidência [7].

2.1.2.4 Algoritmo Johnson

O algoritmo de Johnson [10] resolve o problema de APSP, permitindo a eliminação de pesos negativos usando Bellman–Ford tendo como origem um vértice s, que é acrescentado ao grafo para garantir que exista um vértice de origem chegando em todos os outros, com custo 0 para todos os outros nós.

Após executar Bellman–Ford, tendo s como origem, as arestas são atualizadas ficando todas positivas. Sendo possível então a partir deste ponto usar Dijkstra para solucionar o problema do menor caminho. O Dijkstra é então aplicado para cada nó de origem para solucionar o problema de APSP.

A complexidade do algoritmo é de $O(|V||E| + |V|^2 log|V|)$, sendo que o algoritmo leva $O(|V||E|)$ para computar Bellman–Ford e $O(|E| + |V|log|V|)$ para cada nó de origem do algoritmo de Dijkstra (isto caso se use a melhor estrutura para ordenar os vértices que é a Fibonacci Heap). No caso de grafos esparsos, este algoritmo apresenta melhor desempenho em problemas de APSP comparado ao Floyd-Warshall, como explica [11].

2.1.3 Algoritmo Betweenness Centrality(BC)

Na teoria dos grafos, centralidade é uma medida que analisa a importância de um vértice em um grafo. No presente trabalho, os esforços estão na compreensão do algoritmo de centralidade *Betweenness Centrality*.

Para entender o algoritmo utilizaram-se as notações e definições dadas em [12]. Considere $G(V, E)$ um grafo ou digrafo(isto porque a definição é a mesma caso o grafo seja direcionado ou não) com vértices V e arestas E. Denote $\sigma(s, t)$ o número de menores caminhos entre s e t e $\sigma(s, t|v)$ o número de menores caminhos entre s e t que passa por v. No

caso de $s = t$, então $\sigma(s,t)$ = 1 e se $v \epsilon \{s,t\}$, então $\sigma(s,t|v)$ = 0.

Definindo tais notações, o cálculo de Betweenness Centrality para um vértice v, $C_B(v)$, é dado pela Equação 2.1. Nesta equação, o numerador computa a quantidade de menores caminhos entre o vértice s,t que tem o vértice v como intermediário, enquanto o denominador computa a quantidade de menores caminhos existentes entre o vértice s e t. A soma para todos os nós $s,t \in V$ resultará no cálculo final do algoritmo para um vértice v do grafo.

$$C_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{2.1}$$

Agora é possível entender a necessidade de se usar algoritmos SSSP ou APSP para calcular a distância(caminho mínimo) entre dois vértices, pois estes são parte essencial do cálculo o Betweenness Centrality. Ainda vale ressaltar que existem modificações a serem feitas nos algoritmos SSSP que ,por padrão, armazenam(escolhem) apenas uma distância de um nó s a t, mas perceba pela Equação 2.1 que é necessário obter todos os caminhos mínimos de s a t.

Para entender com profundidade o funcionamento do algoritmo antes é necessário definir o conceito de *pair-dependency* definido por [5].

Dado que se sabe a distância entre dois pares de vértices s e t e a quantidade de menores caminhos que existe entre esses dois vértices, a *pair-dependency* de um par de vértices s e t em um vértice intermediário v é dada pela Equação 2.2, que é taxa de caminhos mínimos de s a t que tem v como intermediário, podendo ser um número entre 0 e 1.

$$\delta_{s,t}(v) = \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{2.2}$$

Antes do algoritmo de Brandes [5] ser proposto, o cálculo de BC era sempre dividido em duas partes. A primeira parte consiste em calcular o menor caminho entre todos os pares de vértices podendo usar para isso algoritmos SSSP para cada vértice ou resolver usando APSP. Durante esta etapa é importante fazer a contagem de quantos caminhos mínimos existe para cada par de vértices, podendo-se guardar estes valores em um array bidimensional.

A segunda consiste em somar as *pair-dependencies*. Esta é definida como $\delta_{s,t}(v) = \frac{\sigma(s,t|v)}{\sigma(s,t)}$, que é a taxa de menores caminhos de s a t que tem v como intermediário.

Para se descobrir a *pair-dependency* de um par de vértices s e t em um vértice intermediário v, é necessário verificar primeiramente se v está no caminho mínimo de s a t. Para isso ele deve satisfazer a Equação 2.3 que verifica se a distância de s a v somada com a distância de v a t é igual à distância(caminho mínimo) de s a t. Esta equação é o *Lemma 1* apresentado em [5].

$$d_{G(s,t)} = d_{G(s,v)} + d_{G(v,t)} \tag{2.3}$$

Caso de fato v seja intermediário, então calcula-se a quantidade de caminhos usando a Equação 2.4. Este cálculo é possível, pois o resultado destes valores foi obtido na primeira etapa. Basicamente o cálculo é multiplicar a quantidade de caminhos mínimos de s a v e a quantidade de v a t, o que é intuitivo por analise combinatória.

$$\sigma(s,t|v) = \sigma(s,v)\sigma(v,t) \tag{2.4}$$

Por fim, para obter o valor de BC para um vértice v basta somar as *pair-dependencies* obtidas como mostra a Equação 2.5.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} = \sum_{s \neq v \neq t \in V} \delta_{s,t}(v) \tag{2.5}$$

A abordagem descrita leva a uma complexidade temporal de O($|V^3|$) já que na segunda etapa é necessário olhar para cada par de vértice s e t e para cada nó v intermediário no caminho mínimo entre os pares. Isso resulta nos loops mostrados no Algoritmo 3. Já a complexidade espacial mínima exigida será dada pela matriz de |V| linhas e colunas que guarda os menores caminhos entre os pares dos nós, o que conduz a uma complexidade de O($|V^2|$).

Algorithm 3: Loop para cálculo de BC após resolver APSP

```
1 for each vertex s do do
2     for each vertex t do do
3         for each vertex v do do
4             if d_{G(s,t)} == d_{G(s,v)} + d_{G(v,t)} then
5                 σ(s,t|v) = σ(s,v)σ(v,t)
6                 C_B(v)+ = σ(s,t|v) / σ(s,t)
```

Tendo estudado os algoritmos SSSP e APSP e entendido os conceitos do BC, é possível avaliar algoritmos que propõem melhorias no cálculo de BC,como ocorreu neste trabalho com o algoritmo de Brandes; e sugerir modificações em algoritmos SSSP e APSP para computarem BC de forma eficiente com base em Brandes.

2.1.3.1 BC: Algoritmo de Brandes

O primeiro algoritmo que computa BC a ser analisado é o de Brandes proposto por [5]. Este algoritmo permite o cálculo de BC para cada nó no grafo de forma eficiente por usar uma fórmula recursiva para calcular os valores parciais de BC após cada computação de SSSP. O algoritmo inova o processo de cálculo do BC que anteriormente necessitava terminar o processamento dos menores caminhos para só então computar os valores de BC. Tal fato resulta numa redução de complexidade temporal como será visto a seguir.

O algoritmo de Brandes utiliza algoritmos transversais para computar os menores caminhos. De forma mais especifica o autor relata o uso de Dijkstra para grafos com pesos e algoritmo de busca em largura (BFS) para grafos sem peso. Para entender como obter valores parciais de BC defini-se inicialmente um conjunto de predecessores que é dado pelo conjunto mostrado na Equação 2.6, sendo ω a matriz de adjacência, ou seja, os predecessores de v são o conjunto de vértices u que tem aresta com v e estão no caminho mínimo de s até v.

$$P_s(v) = \{u \in V : \{u, v\} \in E, d_G(s, v) = d_G(s, u) + \omega[u, v]\} \tag{2.6}$$

Para obter todos os menores caminhos entre um par s e v, pode-se usar a Equação 2.7. Esta equação mostra que a quantidade de caminhos de um nó v herda a quantidade de caminhos de seus predecessores. Tal abordagem também poderia ser usada para obter a quantidade de caminhos antes de Brandes ser proposto para os casos em que só existem arestas com pesos positivos.

$$\sigma_{s,v} = \sum_{u \in P_s(v)} \sigma_{s,u} \tag{2.7}$$

O diferencial de Brandes será eliminar a necessidade de se computar todas as *pair-dependencies*. Para chegar nesse resultado defini-se a Equação 2.8. Esta equação ganha o nome de *dependency* no trabalho de [5].

$$\delta_{s*}(v) = \sum_{t \in V} = \delta_{s,t}(v) \tag{2.8}$$

Definido a Equação 2.8 chega-se a Equação 2.9 do Lema 5 definida em [5]. Esta equação diz que se existe exatamente um caminho mínimo de um $s \in V$ para cada $t \in V$, então a *dependency* de s em qualquer $v \in V$ é dada pela Equação 2.9.

$$\delta_{s*}(v) = \sum_{w : v \in P_s(w)} (1 + \delta_{s*}(w)) \tag{2.9}$$

Esta equação não é suficiente para computar o BC, pois pode existir mais de um caminho mínimo entre s e t pela definição de BC. Com isso, Brandes enuncia um teorema que adapta a fórmula em questão para permitir que mais caminhos mínimos possam existir entre pares de vértices. O teorema diz que a dependency de um vértice $s \in V$ em qualquer $v \in V$ obedece a Equação 2.10. Para obter a demonstração completa deste teorema buscar em [5]. Este teorema diz que somente parte das *dependencies* do nó w,que é sucessor de v, serão propagadas para o nó v.

$$\delta_{s*}(v) = \sum_{w:v \in P_s(w)} \frac{\sigma_{s,v}}{\sigma_{s,w}}(1 + \delta_{s*}(w)) \tag{2.10}$$

Com as alterações propostas o BC pode ser calculado em um tempo $O(|V||E| + |V|^2 log|V|)$ para grafos com peso e em $O(|E||V|)$ para grafos sem peso.

Em [5] é mostrado o pseudocódigo do algoritmo para grafos sem pesos, mas o autor garante a possibilidade de implementação para grafos com pesos, podendo-se usar para isso ,por exemplo, o algoritmo de Dijkstra como núcleo ao invés de BFS. Além disso, são criadas estruturas para armazenar a quantidade de caminhos que são contabilizadas por meio da Equação 2.7 e estruturas para armazenar todos os predecessores nos menores caminhos, que são necessários para se computar os valores parciais de BC após a computação de SSSP. É necessário também armazenar a ordem de visita dos nós durante o SSSP, pois na etapa de atualização dos valores parciais de BC as *dependencies* dos nós visitados por último devem ser atualizadas primeiro, podendo para isso usar pilhas para armazenar os nós visitados. Caso a ordem de visita não fosse esta, os valores de *dependencies* seriam propagadas de maneira errada resultando em valores parciais atualizados erroneamente.

2.1.3.2 BC: Algoritmo Johnson modificado

A vantagem deste algoritmo em relação ao proposto por Brandes [5] é permitir o uso de arestas negativas. Também é possível obter os valores parciais de centralidade obedecendo a fórmula de Brandes em cada computação de Dijkstra, lembrando que antes é necessário atualizar as arestas negativas computando Bellman–Ford.

Para obter a medida de centralidade roda-se Dijkstra, tendo cada nó como origem. Sendo que se deve modificar Dijkstra para armazenar todos predecessores dos nós nos menores caminhos entre a origem e o destino, bem como contar a quantidade de caminhos entre esses dois nós para então realizar o cálculo parcial de BC proposto por Brandes. A complexidade do algoritmo continua a mesma de antes das modificações propostas, tendo a mesma complexidade que o algoritmo de Brandes para grafos com pesos, que é de $O(|V||E| + |V|^2 log|V|)$.

2.1.3.3 BC: Algoritmo Floyd–Warshall modificado

É possível usar o algoritmo Floyd-Warshall para cálculo de Betweenness Centrality. O maior problema desta abordagem é que computa-se todos os menores caminhos entre todos os pares de vértices de uma só vez para posteriormente realizar o cálculo completo de BC para cada nó. Com isto, só se consegue usar as abordagens propostas antes do algoritmo de Brandes, que permitem o cálculo de BC somente depois de se completar o APSP. A complexidade temporal deste algoritmo ainda é a pior dentre os algoritmos estudados sendo $O(|V|^3)$.

2.1.4 Estudo sobre FPGAs

Dado que será um FPGA a processar o algoritmo proposto é importante descrever algumas de suas características.

Os FPGAs são dispositivos de computação reconfiguráveis que contêm um grande número de unidades programáveis. Estas unidades são usadas para resolver problemas computacionais específicos como explica [13], como processamento em grafos, processamento de imagens, entre outros.

Outra característica dos FPGAs é que são dispositivos orientados por dados, isto significa que podem processar dados diretamente sem ter que primeiro decodificar as instruções como ocorre em CPUs e GPUs, que são orientados por instruções. Tal feito garante uma eficiência em termos energéticos como observa [13].

Infelizmente a reconfigurabilidade do FPGA acaba por trazer um custo na frequência sendo que esta cai em torno de 3 a 10 vezes em relação as CPUs, mesmo apresentando essa desvantagem os projetos em FPGA ainda podem superar implementações de CPU ao se utilizar de paralelismo massivo como explica [13].

A maneira tradicional de programar um FPGA é usando linguagens de descrição de hardware (HDL). Estas linguagens permitem descrever o comportamento ciclo a ciclo do hardware. Um problema existente nestas linguagens é fato de serem de baixo nível, o que resulta em uma falta de vários conceitos de alto nível usados nas linguagens de programação de software como aponta [13]. Para solucionar tal problema, pode-se gerar o HDL usando uma ferramenta de síntese de alto nível (HLS). Existem diversas ferramenta HLS sendo que a maioria é baseada em C, C++, que usam diretivas para ajudar o desenvolvedor a expressar recursos arquitetônicos como argumentado por [13].

A FPGA disponível para o presente trabalho é um Pynq-z1 (ZYNQ XC7Z020-1CLG400C). A arquitetura do Zynq é dividida entre o sistema de processamento(PS) e a Lógica programável (PL). A primeira parte é composta de um System on a chip(Soc) ARM dual-core Cortex-A9 embutido permitindo a programação em software, enquanto a segunda pode ser programada usando linguagens de descrição de hardware como explica [14].

Outros componentes importantes no PS são o controlador de memória DDR3(*Double Data Rate 3*) com canais de acesso direto a memória (DMA) e as HP (*High Performance*) AXI3 (*Advanced eXtensible Interface 3*) *slave ports* e vários controladores periféricos com suas entradas e saídas multiplexadas para pinos dedicados, que são chamados de pinos MIO (*Multiplexed Input/Output*) .

Neste ponto é interessante discutir o mapeamento que o compilador HLS faz a partir do código em C++ e das diretivas de compilação para gerar a arquitetura RTL. Todas as funções em C++ são mapeadas em blocos na hierarquia RTL sendo que a função top tem seus parâmetros mapeados nas portas de entrada e saída do modelo RTL. BRAMs (Block Random Access Memories), que guardam grande quantidade de dados na FPGA, podem

guardar arrays. Estas observações são feitas por [14].

A comunicação entre o PS e o PL se dá por meio de interfaces que operam de acordo com protocolo AXI4. O compilador HLS (Vivado e Vitis) suporta AXI4-Stream (axis), AXI4-Lite (s_axilite) e o AXI4-Master (m_axi) como bem observa [14]. É interessante analisar os protocolos AXI4 suportados pelo HLS citados a partir dos estudos feitos por [14] dessas interfaces.

Começando pela AXI4-Stream, o artigo enuncia que esta pode transferir fluxos sequenciais de dados sendo que não há limitação no comprimento da rajada (burst). Para realizar tal feito é utilizado o DMA. Esta interface é a mais rápida para transferir dados entre PS e PL.

Já a próxima, a AXI4-Lite, é a mais lenta, sendo usada para sinalizar o início do processamento no PL e PS, para coletar informações de status e para indicar o endereço base na memória externa (DDR) que um dado array deve começar a ser lido pela interface AXI4-Master (m_axi).

Por último, a AXI4-Master garante transferência de dados entre PS e PL mapeados nas High performance (HP) da DDR. Este método permite que seja implementada transferência de dados em modo de rajadas (burst mode), podendo transferir as palavras baseadas em um único endereço mapeado na memória. Em termos de facilidade de programação, esta interface é mais fácil de ser programada do que a AXI4-Stream, porém é mais lenta como observa [14].

2.2 Trabalhos relacionados

Nesta seção buscou-se apontar pesquisas que implementam o algoritmo BC em FPGAs. É sabido que existem alguns trabalhos que conseguem computar BC para *datasets* com milhares de nós, usando para isto processadores robustos combinados com múltiplas *threads* e por vezes GPUs. Tal uso possibilita alcançar grande paralelismo e obter considerável melhoria no tempo de computação do algoritmo, como abordado nos trabalhos [4] e [3]. Isso não poderia ser feito em FPGAs como a Pynq-z1 por suas limitações de recursos hardware. Todavia o objetivo do trabalho está mais alinhado ao fato de conseguir uma implementação satisfatória de um algoritmo pouco analisado em FPGAs do que obter o melhor desempenho temporal deste.

O *survey* mais recente ,em inglês, encontrado que analisa implementações de algoritmos de grafos em FPGA foi [13]. Este *survey* avalia as implementações de algoritmos SSSP, APSP, BFS, entre outros algoritmos. As implementações descritas foram feitas usando diretamente linguagens de descrição de hardware, como VHDL e Verilog, e também usando HLS, sobretudo para algoritmos SSSP. No caso do BC não é listada uma implementação que usa HLS.

Houve grande dificuldade de achar implementações de BC em FPGA que usam HLS até o presente momento, março de 2022. Durante as buscas, usou-se palavras chaves como *Betweenness Centrality*, *graph processing*, *Centrality measures*, FPGA e HLS de forma combinada, tendo as buscas sido realizadas no site do *Institute of Electrical and Electronic Engineers* (IEEE) e no *Google Scholar*. Por conta destas dificuldades, buscou-se trabalhos feitos com algoritmos SSSP implementados em FPGA, já que estes algoritmos estão presentes no processamento de BC e pelo fato de estarem implementados em FPGA podem ser um guia útil para expor ideias e resultados no presente trabalho. Portanto, foram usados os mesmos sites com as palavras chaves sendo *pathfinding*, *graph processing*, FPGA, HLS e Dijkstra de forma combinada. Ao final, após um filtro dos trabalhos encontrados, decidiu-se considerar as implementações em FPGAs de [14] e [15].

Foi analisado o survey [13] para entender alguns desafios e conceitos em FPGA. Este ressalta que processamento de grafos de maneira eficiente em áreas como aprendizado de maquina, ciência computacional, análises em redes sociais e muitas outras se tornou uma necessidade. Aponta também que o tamanho do conjunto de dados, combinado com a natureza irregular do processamento de grafos cria desafios únicos para o tempo de execução e energia consumida. A partir disto, avalia que os FPGAs podem ser uma solução viável para computar algoritmos de grafos, pois criam um hardware especializado no processamento destes, garantindo, por vezes, um melhor consumo enérgico e desempenho temporal, dependendo da arquitetura construída.

Além do survey expor conceitos importantes para os FPGAs é feita uma explicação dos algoritmos computados em grafos e disponibilizadas referências para obter mais detalhes sobre cada algoritmo. Dentre estes algoritmos, os que estão mais ligados à proposta de trabalho são: SSSP, APSP e BC, que já foram explicados em seções anteriores. Ao avaliar os trabalhos em FPGAs listados no survey, não se encontrou nenhuma implementação do algoritmo BC em FPGA usando HLS. Apesar de poucos trabalhos implementando BC em FPGA, existem trabalhos que implementam algoritmos SSSP usando Verilog, VHDL e HLS como mostrado por [13]. Além de implementar os referidos algoritmos também disponibilizam *frameworks* para serem usados por outras pessoas que queiram fazer pesquisas sem ter de começar a fazer o processo do zero, o que pode ser muito custoso para Verilog e VHDL. No presente trabalho, não se recorreu a tais implementações, pois o objetivo extrapola a implementação de algoritmos SSSP.

Entendendo-se alguns aspectos de funcionamento do FPGA, é possível analisar trabalhos que implementam algoritmos SSSP em FPGA.

Começando por [14], este indica que muitos sistemas embarcados, como *smartphones* e *tablets* são equipados com *Multiprocessor system on a chip* (MPSoCs) personalizados internamente, sendo geralmente construídos em torno de arquiteturas ARM *multi-core*. Expõe que fornecedores como Xilinx e Altera possuem microprocessadores ARM embutidos em torno da lógica programável de seus chips reconfiguráveis, permitindo extensão das funções básicas do ARM. Relata que com o HLS é possível traduzir um código em C++ para

um código em HDL,como VHDL ou Verilog. Tal feito faz com que seja fácil gerar novos protótipos de hardware eficientes e que permitem paralelismo.

A partir destes argumentos, o artigo propõe quatro co-processadores de algoritmos que encontram caminhos mínimos, sendo estes adequados para FPGAs com microprocessadores ARM, que executa a parte de processamento menos intensiva, enquanto o IP no PL executa a demanda de alto desempenho e consumo energético como observado por [14]. O objetivo principal do trabalho é verificar a eficiência e viabilidade da arquitetura RTL construída usando o compilador Vivado HLS, levando em conta que existe a possibilidade de que mais de um caminho mínimo possa ser achado em paralelo. Ao final, analisa a melhora no tempo de inferência desses algoritmos e do consumo energético em relação a um processamento em ARM sem uso de co-processadores.

Para realizar os objetivos cada co-processador de caminho mínimo é projetado usando o Vivado HLS. Focando-se na implementação de Dijkstra em C++, os autores criaram uma fila de prioridade que permite a inserção de elementos com e sem prioridade, ou seja, permite que os elementos sejam ordenados na fila no momento que são inseridos com base em sua prioridade(que em Dijkstra é o custo entre nós) ou não.

É fato que os argumentos da função top (superior) em C++ são traduzidos em portas de E/S RTL. A partir disso, para obter resultados promissores na configuração das interfaces de comunicação entre PS-PL, os autores fazem um estudo dos protocolos AXI4 e selecionam AXI4-Master (m_axi) para transferir a matriz de adjacência do grafo e o os caminhos obtidos como resultado do algoritmo. Tal interface é utilizada, pois permite implementar transferência de dados em modo rajada(*burst*), que possibilita que até 256 palavras de dados sejam transferidas no FPGA em questão com base em um único endereço mapeado na memória como explica [14]. O artigo ainda relata que o protocolo mais rápido é o AXI4-Stream, que permite transferências sequenciais de *Stream* sem limitações no tamanho da rajada(*burst*), no entanto o artigo não utiliza tal protocolo, pois usa DMA, o que torna a programação em PS mais complexa.

Para o correto funcionamento do modo rajada (*burst*), utiliza-se a função *memcpy*, que permite que seus parâmetros sejam escritos ou lidos na memória em modo rajada (*burst*), e diretivas de compilação que expressam o modo como as interfaces devem operar. O resultado final do algoritmo também é transferido usando a interface m_axi, enquanto que outros parâmetros como os próprios endereços, número de nós, nó de origem e nó de destino são transferidos usando a interface AXI-4-Lite. Como último detalhe de escolha dos autores, vale citar que limitaram a quantidade de nós nos grafos para 100, a fim de garantir que fosse possível usar mais de um acelerador ao mesmo tempo no FPGA.

Os resultados foram obtidos usando um FPGA Xilinx Zynq-7000 (XC7Z010-1CLG400C). O que mais importa para o presente trabalho é o resultado obtido ao computar o algoritmo de Dijkstra, pois é o núcleo do algoritmo BC. Avaliando os resultados, o algoritmo de Dijkstra teve uma melhoria de 1,29 x em relação a execução somente em ARM.

Ao incluir mais co-processadores, este desempenho melhorou ainda mais, sendo que para 2 co-processadores foi de 2,57 x e para 3 foi de 8,82 x. Avaliando os resultados de consumo de energia, o PS foi a parte que teve maior consumo, com valores de 1,561 W para todos os algoritmos aproximadamente, enquanto o PL teve valores de menos de 1/4 em relação ao PS.

O segundo trabalho [15] propõe a implementação do algoritmo de Dijkstra usando modelo de arquitetura que permite que estruturas de dados complexas sejam desacopladas do algoritmo, usando para isso interfaces insensível à latência. Isto permite uma execução sobreposta do algoritmo e do método da estrutura de dados para facilitar o pipeline e a paralelização no código pelo HLS.

O trabalho relata que, para melhorar a precisão de um algoritmo, os engenheiros de software separam o programa entre algoritmos e estruturas de dados, sendo que tais programas normalmente estão disponíveis em bibliotecas. Estas buscam atingir resultados de alta performance e proporcionar códigos de fácil reutilização. No entanto, não seria possível usar tais programas em FPGA, pois as ferramentes como HLS focam em algoritmos dominados por implementações que utilizam estruturas de dados primitivas como arrays de tamanho fixo e filas.

Outro ponto observado é que estruturas de dados complexas como filas de prioridade, árvores binárias, entre outras, têm métodos (funções) com dependências de dados e acessos irregulares a memória, o que dificulta o HLS de conseguir gerar um hardware otimizado como explica [15]. Para exemplificar isto, o estudo aborda alguns métodos de árvores binárias como ,por exemplo, o de inserir um elemento na árvore, o que precisa ser feito de forma a permitir somente uma única operação na árvore por vez, ou seja, as modificações na estrutura de dados devem ser feitas de forma serial.

Mesmo assim, as estruturas primitivas podem ser otimizadas a partir de diretivas como pipeline e *unrolling* pelo HLS. O problema é que para estruturas de dados complexas como *heap*, *hash tables*, filas de prioridade e árvores, que geralmente contêm métodos com dependência de dados e acessos irregulares na memória, as otimizações podem não melhorar muito o *hardware* como observa [15]. Para solucionar tal problema implementa-se uma arquitetura que permite desacoplar o método de estrutura de dados do algoritmo.

O trabalho pega emprestado a nomenclatura de programação orientada a objetos para nomear métodos que mudam estruturas de dados de forma serial, chamando-os de métodos mutantes (*Mutator Methods*) e métodos que podem ser executados em paralelos e fora de ordem como métodos de acesso (*Accessor Methods*). No trabalho, métodos mutantes são aqueles que criam dependências de memória, como Escrita após leitura (WAR) e Leitura após escrita (RAW) .

Para melhorar a síntese de estruturas de dados complexas, o trabalho propõe a arquitetura de um SCU (*Specialized Container Unit*) , um template de arquitetura associado, que é composto por diversos módulos conectados usando interfaces baseadas em mensagens in-

sensíveis à latência. A imagem da arquitetura pode ser consultada em [15]. A arquitetura do SCU contém um despachante, que é responsável por receber chamadas de métodos do algoritmo na forma de mensagens que contém códigos de operação para indicar o método requisitado e o valor dos argumentos baseado no tipo do método. O despachante decodifica a mensagem e despacha o trabalho para uma SMU (*Specialized Method Unit*) que executa os métodos de acesso nos A-SMUs (SMU de acessos) ou mutantes nos M-SMUs (SMU mutantes), ou seja, em dois canais separados, um para cada tipo de método. O trabalho ressalta que cada método mutante é sintetizado de forma a criar uma pista única para o M-SMU,ou seja, só existe a execução de um M-SMU por vez, pois modificações simultâneas em métodos mutantes podem gerar inconsistências no algoritmo. Por último, a arquitetura tem um coletor, que é responsável por coletar o resultado das A-SMUs e da M-SMU em operação.

O resultado destas otimizações para Dijkstra ocorre no loop de atualização dos vizinhos do nó que acaba de ser visitado. Cada vizinho v de um nó visitado u tentará ter sua distância atualizada. Caso sua distância seja atualizada, v será colocado na fila de prioridade usando o método *push*. No caso da estrutura não estar desacoplada, o algoritmo precisa parar toda vez que chama o método *push* e no sentido contrário a unidade do método fica sempre ociosa, enquanto o algoritmo roda como bem observado por [15]. Ao usar o desacoplamento, [15] aproveita o fato de que o método push não retorna nenhum valor para que, após receber a requisição, o despachador retorne uma resposta ao algoritmo ao mesmo tempo em que ativa a M-SMU apropriada. Tal feito permite que o algoritmo proceda para a próxima iteração, que é executada ao mesmo tempo que o método *push* da iteração anterior.

Após descrever as vantagens de se usar a SCU, o trabalho indica que esta pode ser sintetizada de forma automática, usando transformações no código combinadas com geradores de hardware parametrizados. Para isto, é necessário identificar no código métodos mutantes e de acesso, o que no caso de Dijkstra já foi revelado(lembre do exemplo com a função *push*). Posteriormente, deve-se retirar a parte do código fonte dos métodos substituindo por instruções de envio e recebimento de mensagens do SCU. O corpo do método, que foi retirado para se dar lugar as instruções, será então usado para criar funções autônomas que podem adicionar módulos de hardware, pois ,como foi dito, o SCU é um template. Com isto o compilador HLS cria os recursos necessários de hardware de forma automática a partir das instruções passadas pelo programador para implementar as funções.

Os resultados de [15] foram obtidos usando um FPGA Xilinx Virtex-7. O objetivo do trabalho foi comparar o resultado entre implementações com estruturas desacopladas e acopladas. Para a implementação usando desacoplamento, a síntese ocorre separadamente entre algoritmo e a estrutura de dados, os quais ,posteriormente, são compostos em um único RTL *Wrapper*. Para o caso do algoritmo de Dijkstra ,que usa uma fila de prioridade implementada como uma árvore binária baseada em array, o SCU conseguiu um aumento de velocidade de 1,82 x, sendo necessário usar 1,5x mais BRAMs. Outros recursos como *Flip-Flops* (FF) e *LookUp Tables* (LUTs) não tiveram aumentos tão significativos, para FF foi de 0,96 x e para LUTs de 1,17x.

Os resultados mostram que, de fato, é possível acelerar o algoritmo de Dijkstra realizando desacoplamento em estruturas complexas, permitindo otimização do tempo. Tal otimização ocorre por conta da continuidade do algoritmo em paralelo com o processamento da fila de prioridade, tomando-se o cuidado para não violar operações sequenciais no processo. O trabalho [15] realmente resolve um problema interessante. No entanto, como o objetivo do presente trabalho é implementar um algoritmo ainda mais complexo em FPGA, e levando-se em conta a pequena experiência que se tem em projetos deste tipo, buscou-se uma implementação convencional, sem desacoplamento.

Capítulo 3

Metodologia

3.1 Escolha do algoritmo

Dentre todos os algoritmos descritos até o momento no trabalho, o algoritmo de Brandes possui as melhores **complexidades de tempo** tanto para grafos com peso como para grafos sem peso, bastando para isso analisar a complexidade dos algoritmos descritos. A complexidade mais próximo a de Brandes é a do algoritmo de Johnson (modificado para computar BC) que pode chegar a mesma complexidade de Brandes para grafos com pesos.

Tal melhoria na complexidade vem do fato do algoritmo permitir o cálculo de BC para cada nó no grafo de forma eficiente, pois usa uma fórmula recursiva para calcular os valores parciais de BC para cada nó em cada computação de Dijkstra. Também apresenta uma vantagem expressiva no uso da **memória** por não necessitar guardar todos os menores caminhos entre cada par de nós simultaneamente diferente de outros algoritmos como o Floyd-Warshall.

Em casos de APSP, o algoritmo Floyd-Warshall é tão mais eficiente quanto mais denso for o grafo, enquanto o algoritmo de Johnson é mais eficiente para grafos esparsos, sendo que uma vantagem comum entre os dois é que podem incluir pesos negativo no calculo do BC. No presente trabalho, considera-se mais vantajosa implementações que beneficiem grafos esparsos, pois é difícil existir grafos completos (ou quase completos) nas áreas que motivaram o trabalho, como nas redes de computadores ou redes sociais.

Os algoritmos de menor caminho com melhor complexidade de tempo para grafos esparsos são o algoritmo de Brandes e Johnson, pela análise de suas complexidades temporais, sendo ambos $O(|E||V| + |V^2|log|V|)$. O que faz ser mais interessante optar por implementar Brandes é o suporte em outros trabalhos e a **simplicidade**, já que Brandes utiliza apenas um algoritmo SSSP em seu núcleo, diferente de Johnson que usa dois algoritmos SSSP (Bellman-Ford e Dijkstra). Vale ressaltar que as complexidades definidas estão levando em conta o uso da estrutura Fibonacci Heap na implementação de Dijkstra. Para obter mais detalhes sobre a complexidade temporal, sobretudo no Dijkstra, que está dentro dos algoritmos

citados, consultar [9].

A partir destas análises, decidiu-se implementar o algoritmo de Brandes, tendo Dijkstra como núcleo para permitir cálculos em grafos com peso, o que torna o problema mais abrangente. Tal escolha foi feita, pois Brandes apresenta resultados melhores em termos de complexidade temporal acompanhada também de maior facilidade de implementação em relação a Johnson, que apresenta mesma complexidade, mas implementa dois algoritmos SSSP.

3.2 Pseudocódigos do algoritmo de Brandes

Escolhido o algoritmo, foram feitos pseudocódigos para auxiliar na implementação deste. O Algoritmo 4 mostra o pseudocódigo de Dijkstra modificado, enquanto o Algoritmo 5 mostra o pseudocódigo de Brandes tendo Dijkstra como núcleo.

É necessário modificar o Dijkstra de forma que este guarde mais de um caminho de menor custo entre uma origem e um destino,fato notado no Algoritmo 4 nas linhas 19 e 23 no array **preds**. Além disto, é necessário guardar a quantidade (valor numérico) de caminhos mínimos existentes entre uma origem e cada destino, para posterior calculo do BC, o que está nas linhas 21 e 24 do Algoritmo 4 no array **qtdPaths**.

A cada vez que se computa o Dijkstra, o seu resultado será entregue ao algoritmo de BC para serem computados seus valores parciais. No Algoritmo 5 ,na linha 6, é possível verificar a presença de uma chamada do Algoritmo 4, enquanto a atualização do valor parcial de Dijkstra está na linha 14 do Algoritmo 5.

3.3 Implementação e comparações dos resultados com a biblioteca Igraph

Após definir os pseudocódigos foi feita a implementação em Python do algoritmo, buscando validar as alterações propostas nos pseudocódigos. Para isso, comparou-se o algoritmo feito em Python com o resultado da implementação de BC da biblioteca Igraph [16] em grafos de 1000 nós, gerados de forma aleatória.

Para medir a proximidade entre os vetores que contêm os valores de BC de cada nó gerados no Igraph e os gerados em Python, foi computada a distância euclidiana entre eles, encontrando-se distâncias da ordem de 10^{-19}, ou seja, a diferença é quase desprezível.

Algorithm 4: dijkstraModified

Input: (w := matriz de adjacência), ($source$:= nó raiz)
Output: $dist$, $pred$, $qtdPaths$, $visitedNodes$

```
1  Declare dist, pred, qtdPaths, visitedNodes
2  for each vertex v ∈ V do
3      visitedNodes[v] ← UNDETERMINED
4      dist[v] ← INFINITY
5      for all pred[v] ← UNDETERMINED
6      qtdPaths[v] ← 0
7  qtdPaths[source] ← 1
8  dist[source] ← 0
9  pred[source] append source
10 for iteration in range |V| do
11     u ← vertex not in visitedNodes with min dist[u]
12     if dist[u] is INFINITY then
13         break
14     append u on visitedNodes
15     for each neighbor v of u do
16         alt ← dist[u] + w[u, v]
17         if alt < dist[v] then
18             dist[v] ← alt
19             for all pred[v] ← UNDETERMINED
20             pred[v] append u
21             qtdPaths[v] ← qtdPaths[u]
22         else if alt == dist[v] then
23             pred[v] append u
24             qtdPaths[v]+ = qtdPaths[u]
25 return dist, pred, qtdPaths, visitedNodes
```

Algorithm 5: BrandesWeighted

Input: (w := matriz de adjacência)
Output: $betwenness$

```
1  Declare betwenness
2  for each vertex v ∈ V do
3      betwenness[v] ← 0
4  for each vertex s ∈ V do
5      source ← s
6      dijkstraModified(w, source)
7      from above we obtain dist, pred, qtdPaths, visitedNodes
8      for each vertex v ∈ V do
9          δ[v] ← 0
10     lastPos = len(visitedNodes) − 1
11     for indice in range len(visitedNodes) do
12         son ← visitedNodes[lastPos − indice]
13         for each father on pred[son] do
14             δ[father]+ = (1 + δ[son]) ∗ (qtdPaths[father]/qtdPaths[son])
15         if son! = source then
16             betwenness[son]+ = δ[son]
17 return betwenness
```

3.4 Implementação no FPGA

Tendo obtido resultados positivos na implementação em Python, busca-se implementar o algoritmo no FPGA Pynq-Z1. Em consequência disso, é necessário usar uma ferramenta de síntese de alto nível (HLS) para facilitar a implementação que será no Vitis HLS. Como já dito, a ferramenta HLS recebe em sua entrada um código em C++ para então gerar o código HDL correspondente.

É importante buscar uma implementação com foco em paralelismo segundo os paradigmas que constam na documentação do Vitis HLS em [17] para conseguir obter um desempenho temporal adequado . O manual insiste que as tarefas de programação devem prioritariamente ser feitas em *pipeline*, tomando-se cuidado também com o uso da memória que é escassa na maioria dos FPGAs. Também sugere a utilização de um modelo produtor consumidor dos dados, pois enquanto dados estão sendo produzidos, outros podem ser consumidos ao mesmo tempo. Claro que, por vezes, seguir todos esses procedimentos pode tornar a tarefa em nível de programação mais complexa e ,por vezes, exigir mais tempo e experiência do programador. Devido ao tempo curto disponível para implementar o projeto e a falta de conhecimento prévio da ferramenta HLS, buscou-se realizar implementações mais simples em nível de programação, buscando sempre que possível evitar situações que prejudiquem o HLS de gerar paralelismo. Adicionalmente, tentou-se economizar o máximo possível o espaço em memória, definindo o tamanho das variáveis a nível de bit.

Outra aspecto importante a ser abordado é que um FPGA precisa de um projeto autocontido especificando todos os recursos necessários para a implementação do hardware. Portanto, não se pode fazer chamadas de sistema que gerenciam a alocação de memória. Com isso, as *structs* definidas em C++ devem ter tamanhos fixos em memória para que a implementação possa ser sintetizada com sucesso. Isto foi feito utilizando constantes numéricas para definir o tamanho dos *arrays* contidos nas *structs*, expressando a capacidade máxima de armazenamento dos *arrays*.

Antes de realizar a implementação em C++ foi feito um mapeamento de todas as estruturas necessárias a serem utilizadas no algoritmo. Vale ressaltar que as estruturas devem ter tamanho fixo, pois o FPGA não consegue alocar memória de forma dinâmica, como já foi explicado.

Para extrair o nó com menor distância, bem como ordenar de forma eficiente as distâncias em cada iteração do algoritmo de Dijkstra, é implementada uma fila de prioridade. A Figura 3.1 mostra a *struct* necessária para a implementação da fila. Perceba que o tamanho máximo desta precisa estar definido. O *array* $data$ guarda a identificação dos nós, $prior$ guarda o custo do nó em relação ao nó de origem. Com isto uma dada posição i na fila contém a identificação do nó,$data[i]$, e o custo do nó até a origem,$prior[i]$. O *array* $visitedNodes$ é necessário para guardar a ordem em que os nós foram visitados no algoritmo de Dijkstra, pois tal ordem será necessária para calcular o BC na fase de computação dos valores parciais. Já as variáveis $size$ e $size\_visited$ são usadas somente para controlar o tamanho atual dos *arrays*, que devem ser sempre menor que o tamanho limite MAX_LIST_SIZE.

```
#define MAX_LIST_SIZE 250

typedef struct
{
    uint_p size;
    uint_p size_visited;
    uint_p data[MAX_LIST_SIZE];
    int_w prior[MAX_LIST_SIZE];
    uint_p VisitedNodes[MAX_LIST_SIZE];
} list_t;
```

Figura 3.1: Struct da fila de prioridade em C++

Para reduzir a complexidade das operações na fila de prioridade pode-se usar uma *Heap*. A que apresenta mais eficiência é a *Fibonacci* levando a uma complexidade temporal de $O(|E| + |V|log(|V|))$, mas apresenta elevado grau de dificuldade de se implementar sem funções recursivas e por conta das limitações com ponteiros em FPGA.

Outro método possível é a utilização de uma *Binary Heap* (árvore binária). Mesmo não sendo a melhor das *Heaps* para o problema, consegue reduzir a complexidade temporal para $O(|E|log(|V|))$. No caso de grafos esparsos ou de tamanho pequeno, o desempenho da implementação por árvore binária pode até ser melhor que para *Fibonacci Heap*, em virtude

da sobrecarga computacional, que deve se tornar mais significativa quando o tamanho do grafo ou densidade do grafo é pequena como explica [9].

Outro fator que contribui para usar a árvore binária é o fato de poder ser facilmente implementada usando *arrays*. No presente algoritmo, pensa-se em uma árvore binária completa com propriedade de *heap* mínima ,isto é, todos os níveis , exceto o último, estão cheios, sendo que os nós do último nível estão mais a esquerda possível. Além disso, um pai tem sempre valor de distância menor ou igual ao de seu(s) filho(s), para obter em cada iteração do algoritmo o nó com a distância mínima em relação a origem. Nestes casos, para se obter a relação entre os nós pais e filhos na árvore basta considerar que um dado nó na posição i do *array*, n[i], tem seu filho esquerdo como n[2i + 1] e o direto n[2i+2] e por último seu pai como n[(i-1)/2]. Isto é o suficiente para realizar as comparações na árvore.

Com base nos critérios discutidos, utiliza-se como solução uma árvore binária completa para ordenar a fila de prioridade com as operações *extract_min* para extrair o nó com distância mínima; *update_key* para atualizar a distância de um nó; *insert_key* para inserir um novo elemento na árvore, e *delete_min* para remover o nó mínimo.

É fato que alguns procedimentos não podem ser feitos em paralelo em uma fila de prioridade, ao menos não de forma trivial. No caso de Dijkstra usando filas de prioridade, a fase de remoção do nó com menor custo e adição ou atualização de outros nós na fila não é algo simples de ser feito em paralelo.

É importante notar também, que o HLS busca otimizações que permitem operações em paralelo e em *pipeline*. Com isso, dependências em iterações anteriores de um *loop* podem ser vistas com aspecto negativo pelo compilador, dado que impedem que atividades sejam feitas em *pipeline* e em paralelo.

No caso de uma árvore binária, existem *loops* em seus métodos que possuem dependências com iterações anteriores, pois um filho na árvore precisa verificar de forma sequencial se é maior ou menor do que um pai para assim realizar o *swap*, que é a troca de lugar com o pai ou filho(para subir ou descer na hierarquia da árvore), repetindo este procedimento de *swap* até encontrar sua posição na árvore.

Se outra operação de modificação na árvore fosse requisitada, durante uma já em operação, possivelmente ocorreriam problemas na ordenação dos elementos dessa árvore, caso não existisse uma barreira que as impedissem de operar juntas. Sendo assim, somente um método da árvore binária pode modificá-la por vez na implementação feita.

Com o intuito de impedir as violações de tempo, que acabam por impedir que as restrições de tempo do circuito sejam atendidas, foi feita uma implementação de fila de prioridade no modo convencional, isto é, sem usar *heaps* (ainda usando os métodos *extract_min*, *update_key*, *insert_key* e *delete_min*), pois reduzir a complexidade da estrutura de dados melhora o processo de síntese no HLS pelos fatos explicados em [15] e já relatados na seção de trabalhos relacionados. Além disso, ao realizar duas implementações é possível compará-las e verificar a melhoria de uma em relação a outra.

Esta nova implementação compara elemento a elemento até encontrar a posição com prioridade adequada onde se deve inserir o elemento. Mesmo sendo uma estrutura mais simples ainda ocorrem dependências em *loops* no momento de se rearranjar os *arrays* $data$ e $prior$ para que um novo elemento seja inserido na posição correspondente.

Como já foi dito, o Dijkstra é apenas o núcleo do algoritmo BC. Então também foi criada uma estrutura para guardar informações a respeito de cada iteração de Dijkstra como mostrado na Figura 3.2. A variável src guarda o nó considerado como origem; o *array* qtd_paths conta a quantidade de caminhos que cada destino tem em relação a origem; o *array* $neighbors$ guarda os vizinhos do nó mínimo extraído em cada iteração; o *array* $preds$ guarda os predecessores de cada nó caminho mínimo em relação a origem src; e a $list\_t$ min_heap é a fila de prioridade usada para otimizar a extração do nó mínimo.

Como detalhe, vale ressaltar que não há diferença nas *structs* apresentadas para os casos com ou sem uso de árvore binária para implementar a fila de prioridade. O que fatalmente muda é a programação dos métodos (funções) *extract_min*, *update_key*, *insert_key* e *delete_min*.

```
typedef struct{

    uint_p src;
    fixed_f qtd_paths[N_NODES];
    uint_p neighbors[N_NODES-1];
    uint_p neighbors_size;
    int_p preds[N_NODES*(N_NODES-1)];
    list_t min_heap;

} dijkstra_calc_t;

typedef dijkstra_calc_t * dijkstra_calc;
```

Figura 3.2: *Struct* para guardar informações úteis durante e após computar Dijkstra

É possível que o leitor tenha percebido tipos de dados diferentes nas variáveis das *structs*, isto porque o HLS permite que a precisão dos tipos seja definida pelo próprio usuário através das bibliotecas "ap_int.h"e "ap_fixed.h". Durante a implementação foi criado um arquivo definindo a quantidade de bits ocupados por cada tipo numérico. Os tipos estão definidos conforme a Figura 3.3. Ao definir a quantidade de bits para cada variável é possível reduzir o tamanho ocupado em memória pelas *structs* e com isso o algoritmo pode conter grafos com mais nós.

O tipo *ap_fixed* mostrada na Figura 3.3 permite operações com números fracionários. Para isso destina parte dos seus bits para gerar a parte inteira e outra para gerar a parte fracionária. A precisão atingida pela parte fracionária dependerá da quantidade de bits que foram reservados para ela. A notação da biblioteca é *ap_ufixed* <total_bits,bits_inteiros>. A precisão atingida para a parte fracionária será de $\frac{1}{2^{(total\_bits - bits\_inteiros)}}$.

```
#include "ap_int.h"
#include "ap_fixed.h"


//type for positions
typedef ap_uint<9> uint_p;

//type for weights
typedef ap_int<17> int_adj;

typedef ap_int<26> int_w;

//int type for positions can use -1
typedef ap_int<10> int_p;

//type for bools
typedef ap_uint<1> bool_t;

//type for float in general
typedef ap_ufixed<40,24> fixed_f;
```

Figura 3.3: Tipos arbitrários definidos a nível de Bits no Vitis HLS

Listagem 3.1: Função *top* no Vivado HLS

```
#include "brandes.h"
static graph_t g;

void brandes_algorithm(volatile int_adj *graph_addr, volatile fixed_f *cf_addr, uint_p n_nodes)
{
        #pragma HLS INTERFACE m_axi depth=122500 port=graph_addr offset=slave
        #pragma HLS INTERFACE m_axi depth=350 port=cf_addr offset=slave
        #pragma HLS INTERFACE s_axilite port=graph_addr bundle=AXI_Lite1
        #pragma HLS INTERFACE s_axilite port=cf_addr bundle=AXI_Lite1
        #pragma HLS INTERFACE s_axilite port=n_nodes bundle=AXI_Lite1
        #pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite1

        fixed_f bet[N_NODES];
        fixed_f sig[N_NODES];
        dijkstra_calc_t dij_t;
    memcpy(g.m,(const int_adj*)graph_addr,n_nodes*n_nodes*sizeof(int_adj));
    g.n_nodes = n_nodes;
    clean_bets: for(uint_p c = 0; c < g.n_nodes; c++)
    {
        bet[c] = 0;
    }
    //brandes loop
    src_loop: for(uint_p v = 0; v < g.n_nodes; v++)
    {
        uint_p src_pos = v;
        compute_dijkstra(&dij_t, &g, src_pos);
        //clean sig array
        clean_partials: for(uint_p c = 0; c < g.n_nodes; c++)
        {
             sig[c] = 0;
        }
        int last_pos = dij_t.min_heap.size_visited-1;
        backforward_loop: for(int_p i=last_pos; i >= 0 ; i--)
        {
            uint_p son_pos = dij_t.min_heap.VisitedNodes[i];
            fathers_loop: for(uint_p p=0; p < (g.n_nodes-1); p ++)
            {
                uint_p father_pos = dij_t.preds[ son_pos*(g.n_nodes - 1) + p];
                if(father_pos == -1 )
                {
                    break;// no more fathers
                }
                else
                {
                    sig[father_pos] += (1 + sig[son_pos])*(dij_t.qtd_paths[father_pos] / dij_t.qtd_paths[son_pos]);
                }
            }
            if(son_pos != src_pos)
            {
                bet[son_pos] += sig[son_pos];
            }
        }
    }
    memcpy((fixed_f*)cf_addr, bet, n_nodes*sizeof(fixed_f));
}
```

Definidos as *structs* e tipos, realizou-se a implementação do resto do algoritmo em C++ no Vitis HLS. O Código 3.1 mostra a função top do Vitis HLS, que computa Brandes. Os parâmetros na entrada da função são $graph\_addr$, um *array* que representa a matriz de adjacência do grafo; cf_addr, um *array* contendo o resultado do BC; e por último a quantidade de nós no grafo,n_nodes.

Os argumentos da função *top*, que é a função brandes_algorithm no Código 3.1, são sintetizados em interfaces e portas que agrupam múltiplos sinais para definir o protocolo de comunicação com componentes externos, como o PS e a DDR. Isto está explicado por [18].

Com intuito de otimizar os acessos da arquitetura RTL a memória, é feito o uso da função *memcpy*, que possibilita usar o modo rajada (*burst*). Este modo garante que seja possível ler ou gravar dados usando um único endereço de base seguido por várias amostras de dados sequenciais como explica [19], permitindo maior taxa de transferência de dados entre a arquitetura no PL e a memória, diminuindo os acessos a esta.

Da forma como foram definidas as diretivas de compilação, nas linhas 6 a 11 do Código 3.1, o *host*/processador enviará os valores escalares e *offsets* de endereços para o adaptador s_axilite. Feito isto o adaptador m_axi lerá o endereço inicial do adaptador s_axilite, pois m_axi opera em modo *slave*, e o armazenará em uma fila para então o design começar a ler os dados da memória global. O procedimento mencionado está descrito em [20].

É interessante analisar alguns dos *loops* do código fonte 3.1. Começando pelo src_loop na linha 23, este especifica em sua variável de iteração v o nó de origem, que terá Dijkstra calculado dentro deste *loop*. Após calcular Dijkstra, que está na linha 26 na função $compute\_dijkstra$, outro *loop* aparece na linha 33, o $backforward\_loop$, que acessa os nós visitados em ordem inversa, do último ao primeiro. Dentro deste *loop* existe outro, o $fathers\_loop$ na linha 36, que realiza a atualização dos valores parciais de BC. Chamou-se de filhos (*son*) os nós que são visitados depois dos pais (*fathers*), ou seja, aqueles visitados por último, e portanto são analisados primeiro no $backforward\_loop$, pois este loop realiza o processo de atualização de BC na ordem inversa a que os nós foram visitados.

Terminado o código em C++, foi feita a simulação do código no Vitis HLS para verificar erros na linguagem. Neste processo, também foi feito um arquivo *testbench* para verificar se os resultados para BC são como esperados. Para tanto, foi pre-computado o BC de alguns grafos, usando-se a biblioteca Igraph, e aplicou-se esses grafos, um por vez, como entrada no arquivo *testbench*, que realiza novamente a computação do BC. Caso a saída do *testebench* fosse igual ao resultado pre-computado dos grafos analisados então o código em C++ teria sido implementado de forma correta.

A próxima etapa é rodar a síntese no Vitis HLS para gerar o modelo RTL, a partir do código em C++. Antes, porém, de rodar a síntese percebeu-se que o *loop* que atualiza os valores parciais de BC, $fathers\_loop$, poderia ser otimizado. Por vezes, o compilador pode identificar dependências, que não existem de fato, entre iterações do *loop*, impedindo otimizações. Para contornar isso, colocam-se diretivas de compilação para explicitar ao compilador que

não existe tal dependência. As iterações do *loop* mostrado na Figura 3.4 podem ser feitas em *pipeline*, ou seja, a iteração atual não precisa esperar o resultado da anterior para começar a computar seus valores parciais de BC. Isso resultará em uma melhoria de tempo do algoritmo dado que esperar sempre por multiplicações e somas, para então se computar um novo *loop*, pode ser algo oneroso. Esta melhora foi verifica de forma empírica comparando implementações no FPGA com e sem esta modificação na etapa de síntese.

Outra diretiva de compilação usada será *#pragma HLS pipeline off*, que desabilita o *pipeline* nos *loops* que atualizam a árvore binária e deslocam elementos para cima ou para baixo desta. Isto foi feito para manter a implementação de BC que usa árvore binária dentro das restrições de tempo, pois ,caso contrário, o circuito poderia apresentar comportamento instável por não atender as restrições de tempo. O processo desta otimização será descrito com mais detalhes no capítulo de discussão.

```
fathers_loop: for(uint_p p=0; p < (n_nodes-1); p ++)
{

    #pragma HLS PIPELINE
    #pragma HLS dependence variable=sig type=inter dependent=false

    uint_p father_pos = dij_t->preds[ son_pos*(n_nodes - 1) + p];

    if(father_pos == -1 )
    {
            break;// no more fathers
    }

    else
    {
            sig[father_pos] += (1 + sig[son_pos])*(dij_t->qtd_paths[father_pos] / dij_t->qtd_paths[son_pos]);

    }
}
```

Figura 3.4: Permitindo o *Pipeline* entre as iterações do loop $fathers\_loop$ na atualização de valores parciais de BC.

Terminado alguns processos de otimização, roda-se a Síntese no Vitis HLS. O período do *clock* é de 11 ns para implementação sem *Heap* e de 11,1 ns para implementação com *Heap*. Terminada a síntese, o Vitis HLS mostra relatórios de tempo e estimativas do uso de memória e de outros recursos da arquitetura gerada. Colocou-se aqui as informações dos relatórios que julgou-se mais pertinente. A Figura 3.5 mostra estimativas no uso das BRAMs, FF, LUTs, DSPs (*Digital Signal Processing*) e URAMs (*Ultra Random Access Memory*) para a abordagem clássica (sem *heap*) na fila de prioridade.

Estes resultados de uso de recursos foram obtidos para grafos com 350 nós, pois conseguiu-se alocar memória no FPGA para rodar em grafos de até 350 nós. Com isso, as constantes numéricas, que definem o tamanho dos *arrays* nas *structs*, foram colocadas com valores de 350. Ao fazer isto, estima-se que o FPGA utilizado, PYNQ-Z1, ocupa em torno de 84% de suas BRAMs. Vale ressaltar que a PYNQ-Z1 possui em torno 630KBytes de BRAM e ao considerar que cada uma dessas tem 18Kbits haverá um um total de 280 BRAMs disponíveis como mostrado na Figura 3.5 retirada do software Vitis HLS [21].

Para as mesmas constantes de tamanho nos *arrays*, o uso estimado da memória na abordagem com árvore binária é também de 84 %, como mostrado na Figura 3.6 retirada do Vitis

HLS [21], enquanto o uso de outros recursos, como LUTs e FF, são menos utilizados. Vale ressaltar que diferente das BRAMs estes outros recursos (incluindo os DSPs) tem grande percentual de suas unidades ainda disponíveis para uso no FPGA, não sendo limitadores na escolha da abordagem a ser implementada (com *heap* ou sem ela).

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	165	-
FIFO	-	-	-	-	-
Instance	8	6	12479	14057	-
Memory	228	-	0	0	0
Multiplexer	-	-	-	933	-
Register	-	-	586	-	-
Total	236	6	13065	15155	0
Available	280	220	106400	53200	0
Utilization (%)	84	2	12	28	0

Figura 3.5: Tabela mostrando utilização dos recursos no FPGA usando fila de prioridade clássica

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	165	-
FIFO	-	-	-	-	-
Instance	8	6	11497	13030	-
Memory	228	-	0	0	0
Multiplexer	-	-	-	942	-
Register	-	-	586	-	-
Total	236	6	12083	14137	0
Available	280	220	106400	53200	0
Utilization (%)	84	2	11	26	0

Figura 3.6: Tabela mostrando utilização dos recursos no FPGA usando árvore binaria junto a fila de prioridade

A próxima etapa para exportar o RTL é co-simulação, que objetiva verificar se o RTL tem funcionamento idêntico ao código-fonte em C++. Passando-se do teste de co-simulação, pode-se finalmente exportar o modelo RTL para ser importado pelo Vivado.

A partir de agora foi seguida uma série de procedimentos no Vivado para gerar o arquivo *Bitstream* a partir do modelo RTL exportado, que podem ser encontrados em [22]. Inicialmente, um projeto no Vivado é criado. Após, o modelo RTL do Vitis HLS é importado

e incluído no catálogo de IPs disponíveis no projeto. Então, é criado um design de blocos (BD) e é adicionado a este o PS (zynqQ7) e o PL (que foi nomeado como brandes_alg).

Feito isso, é necessário habilitar a interface de comunicação do módulo PS. Para tanto, as portas axi do IP foram customizadas, habilitando a interface S_AXI HP0 como mostra a Figura 3.7 (obtida ao configurar o IP no software Vivado [23]). Foram também configurados os IPs brandes_alg e zynqQ7 para que os dados operem com comprimento de 64 bits.

Figura 3.7: Interface utilizada na comunicação PS-PL que permite operação em modo *burst*

O Vivado pode agora realizar as conexões entre os IPs de forma automática e após isso precisa também garantir acesso dos IPs a DDR, algo que também faz de forma automática. Após realizar estas ações, o design obtido é mostrado na Fig . 3.8 retirada do software Vivado [23].

Terminada as conexões entre os IPs, ativa-se o comando para gerar um *wrapper* HDL, que faz o BD definido seja o *top-level* na hierarquia do projeto. A partir deste ponto, é necessário realizar os processos para mapear o modelo RTL gerado na FPGA utilizada. Para tanto, são necessárias as etapas de síntese e implementação.

A síntese é o processo que transforma o RTL especificado em uma representação de *gate-level* como explica [24]. Já o processo de implementação transforma a *netlist* lógica vinda do processo de síntese em um design *placed* e *routed* segundo [25], ou seja, realiza as operações para que o circuito seja mapeado no FPGA, atendendo aos requisitos de tempo e potência. Para realizar isto, a implementação é dividida em vários subprocessos para que sejam realizadas otimizações *Post-Place* (após o design ter os recursos lógicos, componentes eletrônicos colocados no espaço do FPGA utilizado) e *Post-Route* (após conectar, usando fios, os componentes já colocados) como explica [25].

Após o termino das etapas de implementação, é possível obter o esquemático do circuito que será carregado no FPGA. A Figura 3.9 (obtida após etapa de implementação do circuito no *software* Vivado [23]) mostra a comparação dos designs dos circuitos a serem carregados usando *Binary Heap*, Figura 3.9(a), e sem uso de *Heap*, Figura 3.9(b).

Os recursos em azul nas Figuras 3.9(a) e 3.9(b) representam parte da lógica programável que está sendo utilizada pelo circuito. As 4 linhas verticais azuis nas figuras indica o uso das BRAMs. Os outros recursos espalhados podem ser multiplexadores, LUTs, decodificadores, entre outros. É difícil dizer apenas olhando para as figuras qual das implementações usou mais recursos do FPGA. Felizmente, o Vivado mostra o uso de recursos como BRAMs, FFs, LUTs e DSPs que serão utilizados pelo FPGA. Tais valores estão na Tabela 3.1. Perceba que os valores divergem dos obtidos nas Figuras 3.5 e 3.6, sendo que recursos como FFs e LUTs foram superestimados pelo Vitis HLS, enquanto os DSPs foram subestimados. Outro fato interessante é que o Vitis HLS havia predito que seriam usados mais FFs e LUTs na imple-

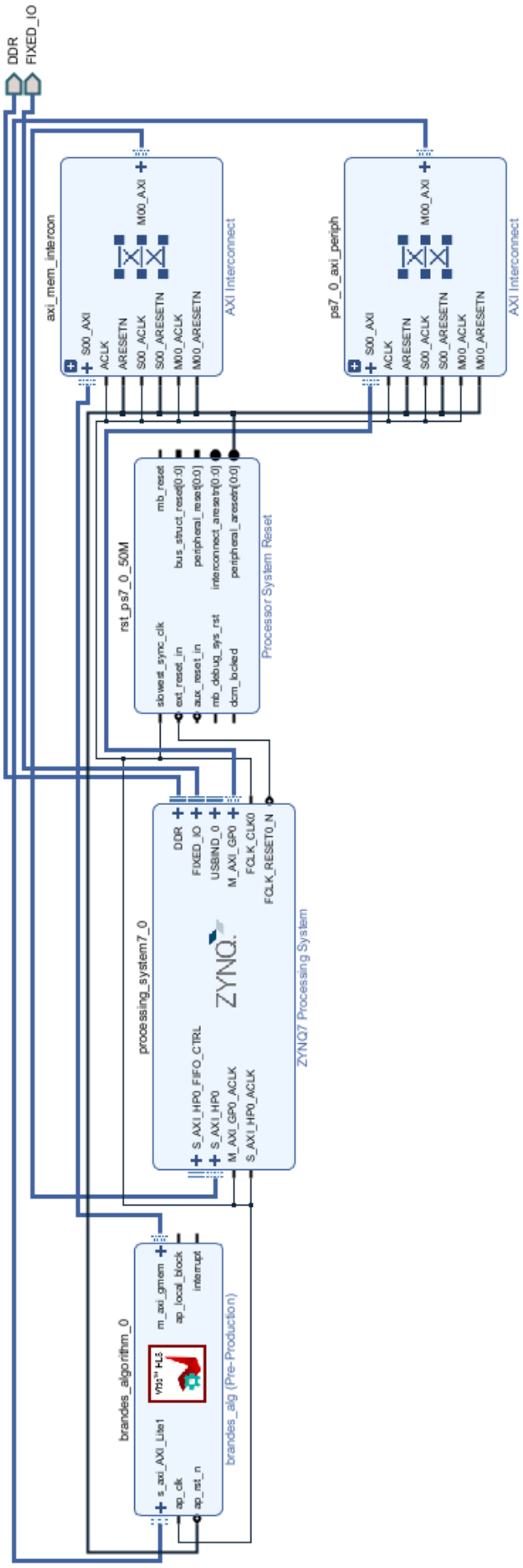
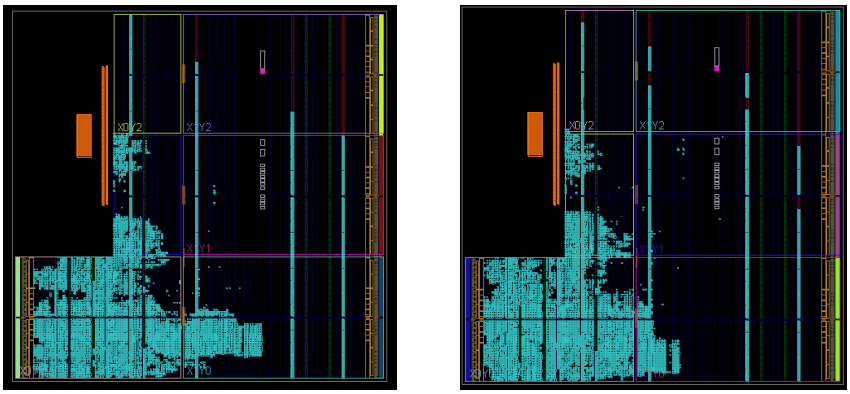


Figura 3.8: Design de blocos entre PS, PL e memória com suas conexões



(a) Esquemático implementação BC com *Binary Heap*

(b) Esquemático implementação BC sem *Binary Heap*

Figura 3.9: Comparação entre as implementações em FPGA

mentação sem *Heap* do que na com *Heap*, mas o que se observou foi que esta implementação usa mais FFs, pórem menos LUTs.

Ao fim da implementação também são obtidas estimativas de potência do circuito, que serão descritas com mais detalhes no capítulo de resultados.

Tabela 3.1: Recursos utilizados no FPGA após implementação no Vivado

	usando *Heap*	**sem *Heap***
BRAM	**116**	**116**
FF	**10658**	**10736**
LUT	**8478**	**8384**
DSP	**14**	**14**

Finalmente, após realizar os processos de síntese e implementação, é possível gerar o arquivo de *Bitstream*, que permite carregar o modelo montado nos blocos lógicos do FPGA. Bastando ser instanciado por uma biblioteca que realiza a extensão do PS no PL. Contudo, antes de gerar o arquivo *Bitstream* dos modelos com e sem *Heap*, buscou-se realizar uma última análise proveniente dos resultados da implementação, referente às restrições de tempo (*time constraints*).

É importante analisar algumas métricas de tempo geradas ao término da implementação para garantir que o circuito gerado realmente possa performar como esperado. No caso da árvore binária (*Binary Heap*), os resultados das métricas de tempo foram como na Figura 3.10, enquanto os da sem *Heap* estão na Figura 3.11 (ambas imagens extraídas do software Vivado [23] após etapa de implementação). Os *clocks* foram escolhidos com base em testes entre períodos de 10ns a 12ns que impedissem valores negativos de *Worst Negative Slack* (WNS) , que é a pior folga de todos os caminhos de tempo para análise de atraso (*delay*) máximo, e *Total Negative Slack* (TNS) , que é a soma de todas as violações de WNS. Caso estas métricas sejam negativas o circuito implementado terá um comportamento imprevisível.

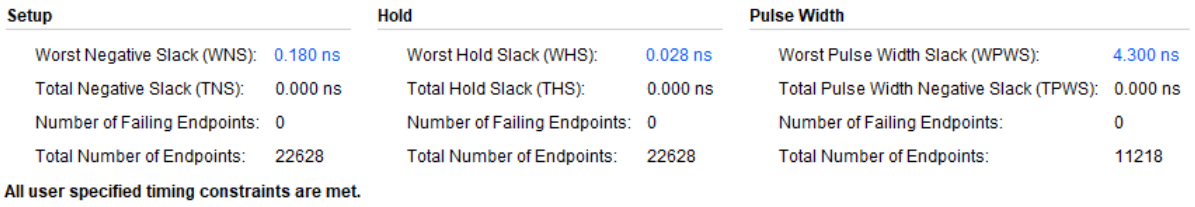


Figura 3.10: Restrições de tempo para implementação usando árvore binária

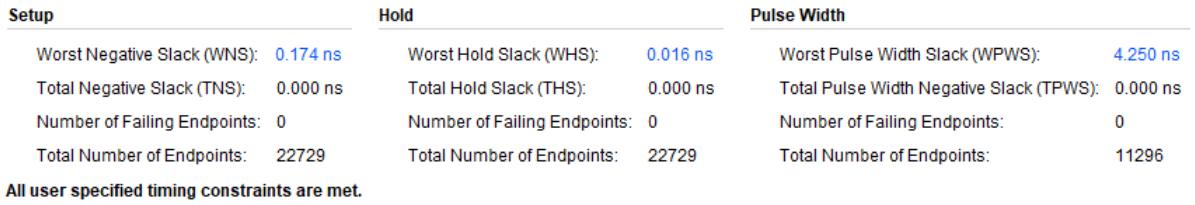


Figura 3.11: Restrições de tempo na abordagem de fila de prioridade sem *Heap*

Ao obter o arquivo *Bitstream* (.bit) é necessário fazer o *download* dos arquivos .bit, .tcl e .hwh gerados pelo Vivado no FPGA. Existem procedimentos que devem ser feitos para conseguir acessar o FPGA remotamente. Antes de tudo, é necessário seguir os passo de instalação da imagem do Linux no cartão SD a ser inserido no FPGA, bem como a conexão dos *jumpers*, que permite o FPGA ser alimentado via USB pelo computador e para conseguir se conectar a internet por meio de sua interface *Ethernet*. Tais procedimentos podem ser encontrados em [26]. Ao seguir os passos propostos deve ser possível acessar remotamente o FPGA por meio do protocolo SSH (*Secure Shell*) .

Com os arquivos importados, é necessário desenvolver um código fonte em Python para realizar a interface entre o módulo PS e PL do FPGA. A biblioteca *Overlays*(com detalhes em [27]), escrita em Python, permite que uma aplicação, programa do PS, seja estendida ao PL, portanto foi feito seu uso. A função em Python para solicitar processamento do PL é descrita no Código 3.2. Esta função recebe uma matriz de adjacência e retorna o resultado de BC. Perceba que para o IP no PL acessar a DDR (a memória SDRAM (*Synchronous Dynamic Random Access Memory*) externa ao PL) é necessário que a memória seja alocada, primeiramente, o que é feito usando a função *allocate*. Com isso, um *array* em Python é alocado em algum lugar da memória virtual e o endereço da memória física deve ser passado para o IP no PL podendo ser acessado em Python através do atributo *physical_device* da função *allocate*, como mostrado nas linhas 12 e 13 do Código 3.2.

Outro ponto importante a se comentar é a respeito da precisão selecionada ao usar a biblioteca ap_fixed mostrada na Figura 3.3. O tipo definido usa 40 bits, sendo 24 para parte inteira e 16 para parte fracionária. Para converter o número retornado pelo IP no PL, que é do tipo ap_fixed, para um ponto flutuante, basta dividir o número por 2^{16}, que nada mais é do que dividir pela precisão como mostrado no Código 3.2 nas linhas 24 a 29.

Terminado estes procedimentos, é possível testar a implementação do circuito no FPGA

e analisar os rendimentos de energia e tempo.

Listagem 3.2: Função em python3 que evoca o PL

```
def compute_brandes(matriz_adj_f):
    n_nodes = math.sqrt(len(matriz_adj_f))
    entrada = allocate(int(n_nodes*n_nodes), "i4")
    saida = allocate(int(n_nodes), "u8")

    overlay = Overlay("brandes.bit")
    brandes = overlay.brandes_algorithm_0.register_map

    entrada[:] = matriz_adj_f

    brandes.graph_addr_1 = entrada.physical_address
    brandes.cf_addr_1 = saida.physical_address
    brandes.n_nodes = int(n_nodes)

    start = time.time()
    brandes.CTRL.AP_START = 1
    while brandes.CTRL.AP_IDLE == 0:
        pass

    total_time = time.time()-start

    saida_float = list()
    precision =  (1/2**16)
    for i in range(int(n_nodes)):
        if saida[i] != 0:
            saida_float.append(saida[i] * precision)
        else:
            saida_float.append(0)
    return {"output": saida_float  , "time": total_time}
```

Capítulo 4

Resultados

Tendo gerado o arquivo *Bitstream* e feito o código em Python para comunicação com o PL, foram feitos testes em grafos gerados de forma randômica entre 5 e 350 nós, pois o FPGA só consegue alocar memória para até 350 nós em um grafo.

Os pesos nos grafos de teste variam entre valores inteiros de 1 a 10. Com isso a maior distancia possível fica sendo 349*10=3490, que pode ocorrer quando um determinado nó precisa passar por todos os outros para chegar a um nó de destino. O que está dentro da quantidade disponível de bits para o tipo int_w, que armazena as distâncias entre os nós, tendo 26 bits para isto, sendo 25 para valores e um bit para dizer se o sinal é positivo ou negativo.

Criou-se uma função capaz de gerar dois tipos de grafos para testar no FPGA. Uma que gera grafos completos, que tem todos os nós conectados entre si, e outra que gera grafos com quantidade aleatória de arestas, permitindo que os nós tenham vizinhanças diferentes.

Na função para gerar grafos com quantidade aleatória de arestas, colocou-se um delimitador de modo que para grafos direcionados a quantidade de arestas fosse no máximo a metade em relação a quantidade total de arestas possíveis, enquanto para grafos não direcionados o limite continua sendo a quantidade máxima de arestas. O cálculo de BC para grafos não direcionados é computado da mesma forma que o para grafos direcionados, bastando para isso considerar que uma aresta sem direção são duas aresta direcionadas com pesos iguais, sendo que ao obter o resultado deve-se dividir os valores de BC de cada nó por 2 como explica [5]. Se o usuário quiser pode normalizar os valores para estarem entre 0 e 1 dividindo os resultados de BC de cada nó pela soma de todos os BCs dos nós.

Para se obter a potência consumida pelo circuito, pode-se utilizar diversas abordagens. O próprio Vivado realiza estimativas de potência em todos os estágios do fluxo. Estes estágios são: *post-synthesis*; *post-placement*; e *post-routing* como descrito em [28]. A precisão da medição melhora ao se modificar as configurações de medição de potência para se adaptar as condições em que o circuito opera, como: *clock*, temperatura ambiente, *junction temperature*(a temperatura do dispositivo em operação), entre outros parâmetros. Também é pos-

sível melhorar a precisão a partir de arquivos SAIF(*Switching Activity Interchange Format*) , que são arquivos gerados durante a simulação no Vivado, que permitem melhor comparação com desempenhos reais do FPGA. Para obter maiores detalhes sobre consumo de potência consulte [28].

A partir disso, os parâmetros para medição de potência foram configurados como mostra na Figura 4.1 (imagem extraída da configuração feita no software Vivado [23]). Não foi fornecido nenhum arquivo SAIF no processo de estimação. O resultado obtido a partir dessas configurações, após o termino das etapas de implementação, foi como mostrado na Figura 4.2 para implementação com fila de prioridade sem *Heap* e como na Figura 4.3 (ambas as imagens extraídas do Vivado [23] após etapa de implementação) para implementação com *Heap*. Lembre que os resultados mostrados são apenas uma aproximação do consumo do circuito.

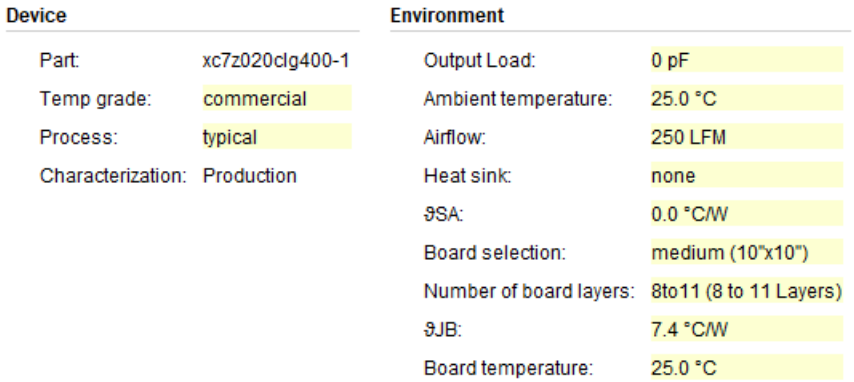


Figura 4.1: Configurações para medições de potência

Perceba dos resultados que o maior consumo de potência no *on-chip* é por conta do PS, que consume 1.256 W nos dois métodos, enquanto que o resto de consumo (*clock*, sinais, BRAMs, entre outros), exceto o *device static*, que é a potência do vazamento do transistor em todos os trilhos de tensão conectados [28], é de 134 mW para o método sem *heap* e de 111 mW usando *heap*, sendo responsável por aproximadamente 1/10 da potência no circuito.

A fim de comparar o desempenho do ARM sozinho com o do mesmo usando os circuitos sintetizados para ajudar no processamento, pensou-se em gerar grafos com vários formatos diferentes. Foram feitos testes em grafos direcionados e não direcionados, completos, que tem todos os nós como vizinhos, e não completos. Além disto, variou-se a quantidade de nós do grafo começando de 70 vértices (nós) até 330. Lembrando que a quantidade máxima de nós permitidas é de 350, pois as *structs* tem *arrays* que foram definidos com este tamanho máximo no código em C++.

Comparando os resultados de tempo do algoritmo usando o co-processador ao usando somente o ARM para a implementação sem *Heap*, foram obtidos os resultados mostrados na Tabela 4.1 na última coluna. Ao usar o circuito para realizar o co-processamento junto ao ARM, o tempo de duração do algoritmo fica entre 1,4 e 1,67 x mais rápido para os grafos

testados.

Testando os mesmos grafos para o co-processador que implementa BC usando *heap*, obteve-se resultados um pouco melhores em relação aos obtidos sem uso de *heap* conseguindo redução de até 5 % no tempo de inferência do algoritmo. Os resultados, neste último caso, poderiam ter sido ainda melhores se não fosse pela dificuldade do HLS de lidar com estruturas de dados complexas, como *heaps*.

O cálculo de BC para grafos não direcionados é feito da mesma forma que para grafos direcionados, fazendo que uma aresta do grafo não direcionado seja mapeada em duas arestas com pesos iguais, no fim convertendo um grafo não direcionado em direcionado, como já explicado anteriormente. A coluna das arestas na Tabela 4.1 no caso de grafos não direcionados indica a quantidade de arestas que o grafo não direcionado tem, mas no momento de se computar BC o número de arestas que existem são como em um grafo direcionado, ou seja, é duas vezes o valor visto na tabela.

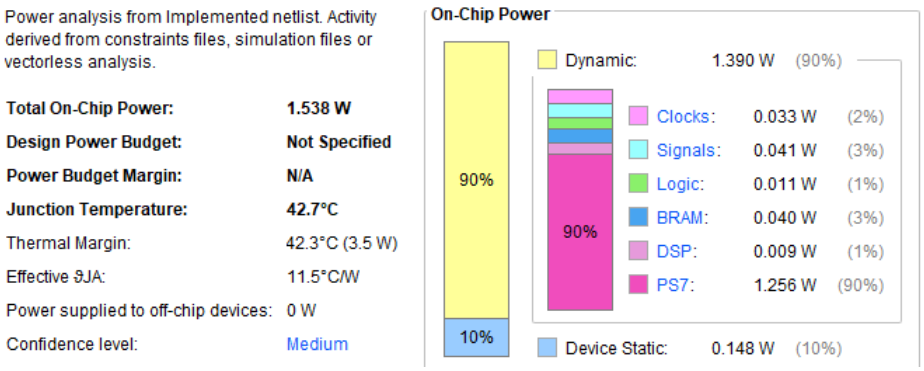


Figura 4.2: Potência estimada na implementação sem *Heap*

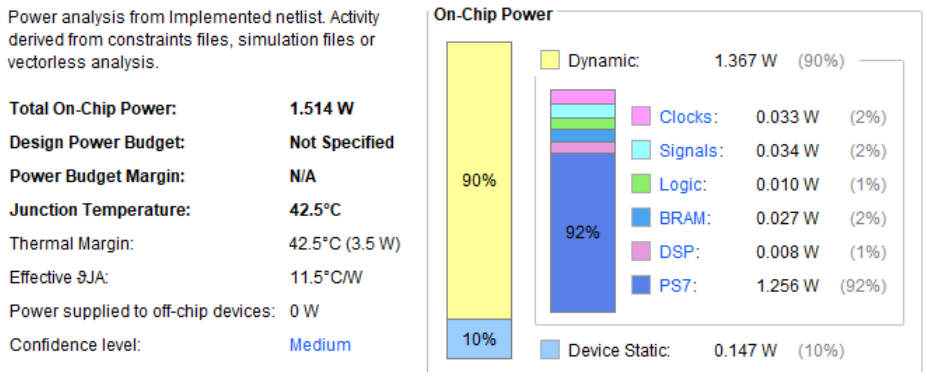


Figura 4.3: Potência estimada na implementação com *Heap*

Tabela 4.1: Melhoria no processamento dos grafos usando coprocessador para caso sem *Heap*

vértices(nós)	arestas	direcionado	completo	melhoria no tempo (sem Heap)
70	575	sim	não	1,67x
135	1268	sim	não	1,48 x
200	3571	sim	não	1,48 x
265	15422	sim	não	1,42 x
330	3731	sim	não	1,45 x
70	4830	sim	sim	1,55 x
135	18090	sim	sim	1,46 x
200	39800	sim	sim	1,42 x
265	69960	sim	sim	1,40 x
330	108570	sim	sim	1,40 x
70	2415	não	sim	1,54 x
135	9045	não	sim	1,46 x
200	19900	não	sim	1,42 x
265	34980	não	sim	1,41 x
330	54285	não	sim	1,40 x
70	503	não	não	1,63 x
135	4547	não	não	1,44 x
200	2245	não	não	1,44 x
265	16593	não	não	1,41 x
330	33145	não	não	1,40 x

Capítulo 5

Discussão

Tendo finalmente colocado os resultados obtidos no trabalho, é interessante discutir alguns pontos importantes. Primeiramente, em relação a quantidade de potência consumida no circuito. Para avaliar se os valores de potência estimados estão dentro de uma margem aceitável para o dispositivo em questão, comparou-se os resultados medidos por [14], que utiliza um FPGA Xilinx da mesma família que o FPGA usada neste trabalho. Ao fazer isto, analisou-se o consumo pelo PS e obteve-se valores de aproximadamente 1,561 W, enquanto no PL foi de 1/4 disto, ou seja, 390,25 mW. Sendo assim, os valores obtidos estão em uma margem de potência aceitável, pois estão na mesma ordem de grandeza dos valores obtidos por um FPGA de características parecidas do mesmo fabricante.

Para colocar os resultados do tempo de inferência do algoritmo neste trabalho, inspirou-se nos resultados mostrados em [14] e [15]. Como já explicado, o primeiro compara quantas vezes o tempo dos algoritmos melhora ao se utilizar co-processadores junto ao ARM em relação ao seu processamento sem estes,ou seja, usando somente o ARM. No segundo trabalho, é verificado quantas vezes o algoritmo foi mais rápido ao usar o método proposto no artigo, que usa desacoplamento de estruturas complexas do algoritmo, em relação ao uso do coprocessador sem aplicar as melhorias propostas.

Ademais, trabalhos que fazem uso de arquiteturas híbridas de GPUs e CPUs colocam resultados usando datasets com grafos de milhares de nós e arestas como em [3]. Todavia, recorrer a tal comparação de resultados é difícil em FPGAs como o utilizado por limitação dos recursos de hardware (principalmente pela falta de memória) e software (pelas limitações na programação em C++). Assim, o presente trabalho teve o foco em mostrar que é possível melhorar o algoritmo BC fazendo com que o coprocessador desenvolvido opere junto ao ARM.

Houve alguns desafios durante o trabalho. Os principais foram entender as diretivas de compilação, os avisos de violação de tempo após a síntese no Vitis HLS e os alertas gerados ao fim da etapa de implementação no Vivado em relação as restrições de tempo. Sobretudo, o mais difícil é saber qual otimização usar e entender o os efeitos positivos ou

negativos que esta pode ter na síntese e implementação do design. Isto exigiu leituras nas documentações online do Vitis HLS e do Vivado (que já tiveram vários fragmentos citados durante o trabalho) e também tempo para segmentar o conhecimento proveniente das fontes citadas no decorrer do trabalho.

Além disso, foram feitos vários testes para se chegar a um modelo que atingisse as restrições de tempo (*time constraints*) após a implementação no Vivado. A seguir, são fornecidos maiores detalhes dos desafios encontrados para satisfazer as restrições de tempo e as soluções usadas para resolvê-los durante o projeto.

Nos primeiros testes feitos no Vitis HLS, a síntese foi feita usando as condições padrões de *clock*,que é de 10ns (100 MHz), e sem nenhuma diretiva de compilação, exceto pelas diretivas nas interfaces de entrada e saída e pela diretiva de *pipeline* identificada no *loop* de computação dos valores parciais de BC. Com isto, no caso da implementação com árvore binária, houve alertas do tipo II *Violations* (que são dependência de hardware que o HLS gera a partir do código fonte do usuário quando o *loop* ou função está com *pipeline* II=1 como explica [29]) e *Timing Violations* (que ocorrem quando um caminho de operações requer mais tempo do que o *clock* disponível [30]) durante o processo de síntese do Vitis HLS e alertas II *Violations* para o caso da fila de prioridade sem *Heap*.

É fato que ambas as implementações apresentam limitações durante o processo de mudanças na fila de prioridade, tanto para atualização, inserção de um elemento na fila quanto para remover o elemento mínimo da fila. Só que a abordagem usando árvore binária apresentou vários II *Violations* e *Timing Violations*, o que pode significar mais riscos no momento de se rodar o algoritmo no FPGA, pois resultará em um circuito com maior *delay* entre os registradores no FPGA, como foi visto após testes exaustivos. Portanto, é recomendável resolver pelo ou menos a maioria das violações na síntese do Vitis HLS.

No capitulo 2, já foram discutidas as dificuldade do compilador HLS implementar de forma satisfatória estruturas complexas, como *Heap*, filas de prioridade, entre outras, ao analisar o trabalho de [15]. Violações de tempo presentes em *loops*,como II *Violations* e *Timing Violations*, dificultam o circuito implementado de atingir as restrições (limitações) de tempo, sobretudo em relação as métricas de *Setup time*, como observado após vários testes. Caso tais métricas não sejam satisfeitas, pode ocorrer metaestabilidade no circuito, como observa [31], o que não é desejável.

No caso da síntese usando árvore binária, constatou-se que as violações ocorreram em *loops* em que existem dependências entre as iterações, precisamente nos momentos em que a prioridade de um elemento na árvore muda ou um novo elemento precisa ser acrescentado ou removido, sendo necessário deslocar elementos para cima ou para baixo na árvore. Estes alertas se dão pelo fato das iterações atuais nos *loops* serem dependentes de iterações anteriores que demoram mais de um ciclo para terem seu resultado lido ou escrito, assim impedindo que uma nova iteração seja computada em cada ciclo como explica [29]. Constatou-se este fato, observando-se o resumo dos resultados após a síntese no Vitis HLS, de forma mais es-

pecifica, avaliando o *scheduler viewer* (visualizador de agenda) dos *loops* que apresentaram violações de tempo. Ressalta-se que nem sempre as violações precisam ser resolvidas e nem todas as violações podem ser resolvidas como explica [32].

A solução encontrada para ajudar a resolver as violações foi explicitar por meio de diretivas de compilação do tipo *#pragma HLS pipeline off*, que o *pipeline* nos *loops* mencionados, para subir ou descer um elemento na *Heap*, fossem desabilitados permitindo a arquitetura construir um circuito que permite um relaxamento nas condições de tempo para computar as iterações dos *loops*. Claro que, em troca, haverá um desempenho pior no que se refere ao tempo de inferência do algoritmo, pois está se desabilitando o *pipeline*. Porém, mesmo usando esta abordagem, não se atingiu os requisitos de tempo necessários, sendo ainda preciso usar outra solução, que foi aumentar o *clock* da implementação de 10ns para 11,1 ns. Ao fazer isto, os alertas II *Violations* e *Time Violations*, já citados, no Vitis HLS somem e o mais importante as restrições de tempo na implementação no Vivado foram atendidas, o que garante correto funcionamento do circuito. O motivo do *clock* ter esse efeito será explicado logo mais.

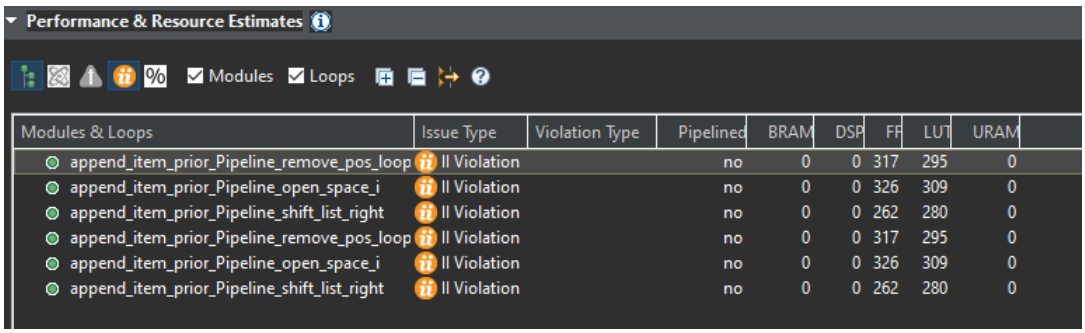
Já para o caso da implementação da fila de prioridade no modo clássico, sem *Heaps* na fila, existiram, também, alertas de II *Violations* para alguns *loops* no código, como mostrado na Figura 5.1 (extraída do software Vitis HLS [21] após etapa de síntese). As violações nesta implementação estão em *loops* com dependência entre as iterações do tipo WAR (*write after read*), não sendo possível puxar uma instrução de escrita antes da variável ter sido lida no *loop* anterior. O Código fonte 5.1 mostra um dos *loops* com dependências WAR entre as iterações.

No Código 5.1, a variável $l->size-1$ é sempre maior ou igual a $index$, sendo este *loop* um dos *loops* usados para deslocar os elementos existentes para inserção de um novo. Como observação, a *struct* do tipo *min_heap* tem no seu nome a palavra *heap*, mas os seus métodos (funções) não implementam uma *heap* de fato, isto porque, como dito anteriormente, as *structs* usadas são as mesmas para implementação com e sem *heap*, não sendo mudado nem mesmo os nomes delas.

Para ajudar a solucionar as violações nesse caso, pensou-se em fazer o mesmo que na abordagem com árvore binária, ou seja, desabilitar o *pipeline* nos *loops* em que existe dependências de dados. Mas, ao realizar alguns testes, percebeu-se ser mais vantajoso aumentar o *clock* de 10ns para 11ns e manter o *pipeline*. Com isto, mesmo sem desabilitar o *pipeline* nos *loops*, foi possível atender as restrições de tempo usando-se um *clock* de 11ns.

Listagem 5.1: Laço dependente

```
1 l = struct do tipo mim_heap
2 for(i = l->size-1 ; i >= index ; i--)
3 {
4     l->data[i+1] = l->data[i];
5     l->prior[i+1] = l->prior[i];
6 }
```



Modules & Loops	Issue Type	Violation Type	Pipelined	BRAM	DSP	FF	LUT	URAM
append_item_prior_Pipeline_remove_pos_loop	II Violation		no	0	0	317	295	0
append_item_prior_Pipeline_open_space_i	II Violation		no	0	0	326	309	0
append_item_prior_Pipeline_shift_list_right	II Violation		no	0	0	262	280	0
append_item_prior_Pipeline_remove_pos_loop	II Violation		no	0	0	317	295	0
append_item_prior_Pipeline_open_space_i	II Violation		no	0	0	326	309	0
append_item_prior_Pipeline_shift_list_right	II Violation		no	0	0	262	280	0

Figura 5.1: Violações de tempo após síntese no Vitis HLS para abordagem sem usar árvore binária.

Como visto, parte da solução encontrada que resolve II *Violations* e *Time Violations*, para atender as limitações de tempo, foi aumentar o período do *clock*, ou seja, diminuir sua frequência, que por padrão é de 100 Mhz no Vitis HLS e no Vivado (período de 10 ns) para ,aproximadamente, 90,91 MHz (11 ns) no caso sem *heap* e de 90,01MHz (11,1 ns) para o caso com *heap*. Tais mudanças contribuíram para sumir com a maioria dos alertas no Vitis HLS e fazer com que métricas como WNS e TNS não tivessem valores negativos no Vivado.

Para entender como o aumento do período do *clock* pôde ajudar a resolver o problema de *Setup Time*, primeiro deve-se entender o que é *Setup Time*, que é o tempo necessário para que a entrada de um *Flip-Flop* esteja estável antes da aresta do *clock*. Isto está melhor explicado na Figura 5.2, retirada de [31], em que t_{su} é o tempo que o dado tem que estar disponível no REG 2 para não ocorrer uma violação de tempo do tipo *Setup Time*.

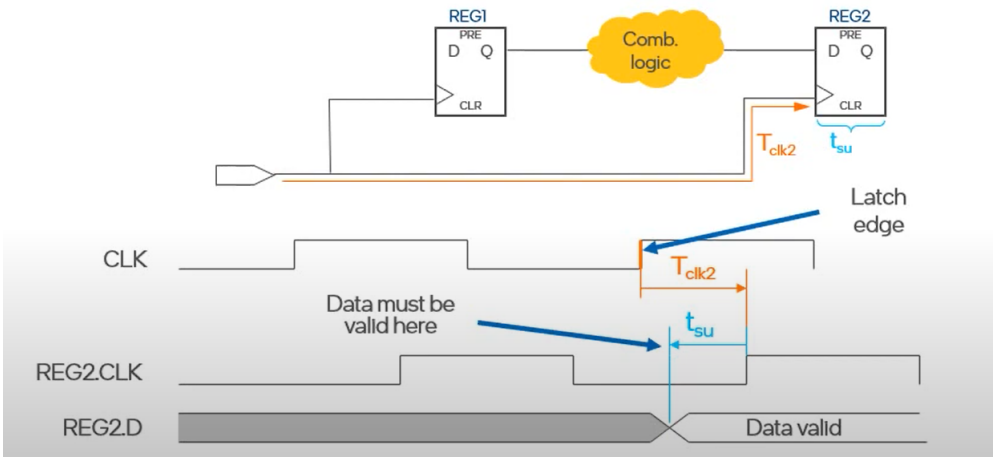


Figura 5.2: *Setup Time* explicado

Caso ocorra uma violação de tempo *Setup Time*, isso significa que o *clock* escolhido foi rápido demais em relação ao *delay* (atraso) de propagação entre os registradores (FFs), de tal forma que os dados não estavam estáveis no registrador (FF) de destino no tempo adequado antes da aresta do *clock*. No caso da arquitetura construída, o Vitis HLS e o Vivado colocam um *clock* padrão de 10ns que ,infelizmente, falhou em atender os requisitos de tempo, ou seja, existem *delays* entre registradores que não atendem ao requisito de chegar no registrador de destino em um tempo adequado antes da aresta do *clock*.

O maior tempo de propagação entre dois registradores no design funciona como um limitador do período do *clock* que se pode escolher, pois quanto menor o período do *clock* (maior a frequência), menor tem de ser o *delay* entre os registradores para que se consiga ter o dado estável antes da aresta do *clock*. Seguindo esta lógica, quanto maior o período do *clock*, maior pode ser o *delay* entre os registradores. Com isto, existem duas opções para atender ao *Setup Time*. Ou se modifica a arquitetura construída pelo Vivado de forma a incluir mais registradores para diminuir o *delay* entre cada par de registradores, ou pode-se aumentar o período do *clock* para permitir *delays* maiores entre os registradores.

Para resolver os problemas de *Setup Time* decidiu-se aumentar o período do *clock* nas implementações para impedir as violações, aumentando para valores superiores a 10ns. Esta abordagem é simples de se fazer em relação a modificar a arquitetura RTL no Vivado, que exige vasta experiência em circuitos digitais, de modo a conseguir modificar a arquitetura existente por uma melhor que realiza as mesmas operações.

Após alguns testes, as implementações com menores *clocks* que puderam atender as restrições de tempo foram de 11,1ns para implementação com *heap* e 11ns para implementação sem ela. Caso o leitor queira saber mais a respeito das restrições de tempo favor consultar [31] e [33].

Vale ressaltar, porém, que simplesmente aumentar o *clock* para o caso da abordagem usando árvore binária não satisfaz seus requisitos de tempo no Vivado. Acredita-se que isto ocorre por conta de sua maior complexidade em relação a uma implementação sem uso de *heap*. Portanto, é necessário unir o aumento do *clock* às diretivas de compilação que impedem o *pipeline* em alguns *loops* do código que tem dependências e acessos irregulares a memória. Para o caso sem utilizar *heap*, não foi preciso desabilitar o *pipeline* para satisfazer as restrições de tempo em seus *loops* sendo somente necessário aumentar o *clock* para 11ns.

Como já foi dito no capítulo 4, a abordagem com *heap* poderia ter resultados ainda mais promissores em relação a abordagem sem *heap*. O fato da abordagem ter uma estrutura mais complexa teve como consequência um *clock* com período maior e *loops* com *pipeline* desabilitado para atingir as restrições de tempo, o que levou a uma queda do desempenho no tempo de inferência. Melhorar ainda mais o desempenho desta abordagem exigiria um conhecimento mais aprofundado em circuitos digitais de modo a modificar a arquitetura implementada no Vivado, propondo uma que seja equivalente e opere melhor do que a sintetizada por este, ou maior conhecimento de programação em FPGA de modo a entender melhor o compilador HLS e com isso estruturar um código que receba melhor tratamento pelo compilador referido. Com isto deixa-se este desafio para outros pesquisadores, com uma recomendação de se utilizar estruturas desacopladas, como foi visto no capítulo 2 na seção de trabalhos relacionados, de modo a permitir maior paralelismo e *pipeline* no FPGA.

Como última análise, também foram verificadas possíveis melhorias por meio de diretivas de compilação como *UNROLL*, que pode ser usada de maneira a permitir que várias iterações de um *loop* sejam processadas ao mesmo tempo. Para isso esta diretiva replica as

estruturas lógicas existentes em cada iteração do *loop*.

No caso do algoritmo de Brandes, tal otimização poderia ser aplicada para o *loop* mais externo, que escolhe a origem do algoritmo de Dijkstra a ser computado. Como é visto em trabalhos como [34], o processamento de cada iteração pode ser feito de forma independente e fora de ordem, fato observado ao se escolher a ordem das origens do SSSP de forma aleatório ou por outros métodos.

O problema de tal otimização é demandar mais recursos como BRAMs, LUTs, entre outros; e já foi visto que o uso das BRAMs para uma única linha de processamento por vez já ocupa em torno de 84 % das BRAMs disponíveis neste FPGA. Nada impede, porém, que sejam feitos testes de *UNROLL* ou *UNROLL factor=x* (neste caso, ao invés de criar uma estrutura lógica para cada iteração, cria-se a quantidade de equivalente a x) em FPGAs com mais recursos.

Ainda existe outra forma de otimizar o tempo do algoritmo, que é reduzir o número de nós tomados como origem em Dijkstra como feito em [34]. Nele são escolhidos nós chamados de *pivots* para terem seus SSSP computados e após isto valores parciais de BC dos nós que tem caminhos até ele atualizados. No entanto, tal feito acaba por reduzir a acurácia do algoritmo. A utilização da estimativa vai depender do nível de acurácia exigida do algoritmo por outros processos. No caso em que a acurácia precisa ser muito alta, pode não ser interessante se utilizar esse tipo de otimização.

Conclusão

Conclui-se com o trabalho que é possível implementar o algoritmo BC, mais especificamente o algoritmo de Brandes, no FPGA Pynq-Z1 e conseguir desempenhos de tempo melhores usando o coprocessador junto ao ARM quando comparado ao resultado obtido por este sozinho.

Constatou-se ainda que o algoritmo sofre melhoria no tempo de processamento dos grafos testados ao se usar árvore binária na implementação da fila de prioridade ao invés de uma implementação da fila sem *heap*, mesmo que não seja uma melhoria tão significativa quanto poderia ter sido pelos motivos discutidos no trabalho.

Avaliou-se também a potência consumida pelos circuitos implementados, e verificou-se que seus valores estão dentro de uma faixa aceitável ao se comparar com implementações em dispositivos semelhantes.

Percebeu-se que o algoritmo BC pode ser melhorado ao se utilizar desacoplamento das estruturas de dados dos algoritmos. Para tanto, é necessário mais experiências de programação em FPGAs a fim de gerar o melhor modelo RTL possível. Outra possível melhoria pode ocorrer ao se utilizar FPGAs com mais recursos, sobretudo em termos de memória (BRAMs).

Por fim, vislumbrou-se a possibilidade de se sacrificar a acurácia do algoritmo a fim de diminuir o tempo de processamento deste. Claro que isto dependerá do nível de precisão que uma dada tarefa exige do algoritmo.

Referências Bibliográficas

[1] CHOWDHURY, S. et al. Botnet detection using graph-based feature clustering. *Journal of Big Data*, Springer, v. 4, n. 1, p. 1–23, 2017.

[2] BORGATTI, S. P. Centrality and network flow. *Social networks*, Elsevier, v. 27, n. 1, p. 55–71, 2005.

[3] SARIYÜCE, A. E. et al. Betweenness centrality on gpus and heterogeneous architectures. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. [S.l.: s.n.], 2013. p. 76–85.

[4] MADDURI, K. et al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: IEEE. *2009 IEEE International Symposium on Parallel & Distributed Processing*. [S.l.], 2009. p. 1–8.

[5] BRANDES, U. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, Taylor & Francis, v. 25, n. 2, p. 163–177, 2001.

[6] FEOFILOFF, P.; KOHAYAKAWA, Y.; WAKABAYASHI, Y. Uma introdução sucinta à teoria dos grafos. 2011.

[7] DROZDEK, A. *Estrutura De Dados E Algoritmos Em C++*. 4ª edição americana - 2ª edição brasileira. ed. SP - Brasil: Cengage Learning, 2017.

[8] FEOFILOFF, P. *Caminhos de custo mínimo*. 2020. Disponivel em: https://www.ime.usp.br/p̃f/algoritmos_para_grafos/, Acesso em: 29 September 2021.

[9] SAUNDERS, S. A comparison of data structures for dijkstra's single source shortest path algorithm. University of Canterbury. Computer Science, 1999.

[10] JOHNSON, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 24, n. 1, p. 1–13, 1977.

[11] TOROSLU, I. H. Improving the floyd-warshall all pairs shortest paths algorithm. *arXiv preprint arXiv:2109.01872*, 2021.

[12] BRANDES, U. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, Elsevier, v. 30, n. 2, p. 136–145, 2008.

[13] BESTA, M. et al. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[14] NERY, A. S.; SENA, A. C.; GUEDES, L. S. Efficient pathfinding co-processors for fpgas. In: IEEE. *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. [S.l.], 2017. p. 97–102.

[15] ZHAO, R. et al. Improving high-level synthesis with decoupled data structure optimization. In: IEEE. *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.], 2016. p. 1–6.

[16] CSARDI, G.; NEPUSZ, T. The igraph software package for complex network research. *InterJournal*, Complex Systems, p. 1695, 2006. Disponível em: <http://igraph.org>.

[17] XILINX. *Optimization Techniques in Vitis HLS*. 2021. Disponivel em: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Optimization-Techniques-in-Vitis-HLS, Acesso em: 09 Março de 2022.

[18] XILINX. *pragma HLS interface*. 2021. Disponivel em: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-interface, Acesso em: 11 Março de 2022.

[19] XILINX. *AXI4 Master Interface*. 2021. Disponivel em: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI4-Master-Interface, Acesso em: 09 Março de 2022.

[20] XILINX. *Offset and Modes of Operation*. 2021. Disponivel em: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Offset-and-Modes-of-Operation, Acesso em: 09 Março de 2022.

[21] XILINX, I. *Vitis Core Development Kit - 2021.2 Full Product Installation*. 2021. Disponivel em: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html, Acesso em: 17 Janeiro de 2022.

[22] PALANIAPPAN, S. *Lab: AXI4Burst Mode*. 2019. Disponivel em: https://pp4fpgas.readthedocs.io/en/latest/axi4.html, Acesso em: 11 Março de 2022.

[23] XILINX, I. *Vivado ML Edition - 2021.2 Full Product Installation*. 2021. Disponivel em: https://www.xilinx.com/support/download.html, Acesso em: 17 Janeiro de 2022.

[24] XILINX. *Vivado Design Suite User Guide: Synthesis*. 2021. Disponivel em: https://docs.xilinx.com/v/u/en-US/ug901-vivado-synthesis, Acesso em: 11 de Março de 2022.

[25] XILINX. *Vivado Design Suite User Guide: Implementation (UG904)*. 2021. Disponivel em: https://docs.xilinx.com/r/en-US/ug904-vivado-implementation/Navigating-Content-by-Design-Processs, Acesso em: 11 Março de 2022.

[26] XILINX. *Python productivity for Zynq(Pynq)*. 2017. Disponivel em: https://pynq.readthedocs.io/en/v2.0/getting_started.html#, Acesso em: 11 Março de 2022.

[27] XILINX. *Overlay Design Methodology*. 2018. Disponivel em: https://pynq.readthedocs.io/en/v2.3/overlay_design_methodology.html, Acesso em: 11 Março de 2022.

[28] XILINX. *Vivado Design Suite User Guide: Power Analysis and Optimization*. [S.l.], 10 2021. Disponível em: <https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx 2021_2/ug907-vivado-power-analysis-optimization.pdf, Acesso em: 11 de Março de 2022>.

[29] XILINX. *Pipeline Constraint Violation*. 2020. Disponível em: https://www.xilinx.com/htmldocs/xilinx2020_1/hls-guidance/200-880.html, Acesso em: 20 de Fevereiro de 2022.

[30] XILINX. *Vitis High-Level Synthesis User Guide*. [S.l.], 12 2021. Disponível em: <https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx 2021_2/ug1399-vitis-hls.pdf, Acesso em: 11 de Março de 2022>.

[31] FPGA, I. *Understanding Timing Analysis in FPGAs*. 2021. Disponivel em: https://www.youtube.com/watch?v=6D-w8mOttnE, Acesso em: 11 Março de 2022.

[32] XILINX. *Using Optimization Techniques*. 2021. Disponivel em: https://xilinx.github.io/Vitis-Tutorials/2020-1/docs/vitis_hls_analysis/optimization_techniques.html, Acesso em: 11 Março de 2022.

[33] MATHWORKS. *Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows*. 2022. Disponivel em: https://www.mathworks.com/help/hdlcoder/ug/resolve-timing-failures-in-ip-core-generation-and-generic-asicfpga-workflows.html, Acesso em: 11 Março de 2022.

[34] BRANDES, U.; PICH, C. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, World Scientific, v. 17, n. 07, p. 2303–2318, 2007.